高等学校计算机教材

汇编语言实用教程

郑阿奇　主　编
吕升旭　编　著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书以 80x86 指令系统为主体，详细介绍汇编语言程序设计，共 10 章。首先复习汇编语言数据运算基础和 80x86 微处理器体系结构，通过一个实例介绍汇编语言程序上机操作过程；然后介绍指令系统和寻址方式、汇编语言源程序的组成和 8086/8088 常用指令，在此基础上，介绍基本程序设计、子程序设计、输入/输出与中断程序设计和简单应用程序设计。本书的主要特点是循序渐进，结构清晰，重点突出，教和学比较方便。

本书可作为高等学校有关专业汇编语言程序设计课程教材或参考书，也可作为自学和微机应用技术人员的参考用书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

汇编语言实用教程 / 郑阿奇主编，吕升旭编著. —北京：电子工业出版社，2011.5
高等学校计算机教材
ISBN 978-7-121-13397-8

Ⅰ. ①汇… Ⅱ. ①郑… ②吕… Ⅲ. ①汇编语言－程序设计－高等学校－教材 Ⅳ. ①TP313

中国版本图书馆 CIP 数据核字（2011）第 078565 号

责任编辑：郝黎明　　特约编辑：田领红
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：20.75　字数：531.2 千字
印　　次：2011 年 5 月第 1 次印刷
印　　数：4 000 册　　定价：35.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

前　言

本书以 80x86 指令系统为主体，详细介绍汇编语言程序设计。主要特点是循序渐进，结构清晰，重点突出，教和学比较方便。各章内容安排基本思路如下。

第 1 章汇编语言运算基础：系统复习汇编语言程序设计使用的数据及其算术和逻辑运算，这些内容在此前的课程已经介绍过，但学生对这部分内容的掌握个体上是有差别的，学习本课程之前需要复习这些内容。读者可根据情况选择学习。

第 2 章微型计算机的体系结构：主要介绍 80x86 微处理器及其操作方式，为 80x86 汇编语言程序设计提供操作基础。

第 3 章汇编语言程序及其操作过程：在介绍机器语言程序的基础上，介绍汇编语言程序。通过一个典型实例介绍机器语言程序上机操作方法，重点是机器语言程序的调试。在此基础上，介绍程序的编辑、编译、连接和运行。

第 4 章汇编语言指令系统和寻址方式：在简单介绍指令系统分类的基础上，介绍指令操作数和为了得到操作数而采用的寻址方式。

第 5 章汇编语言源程序组成：介绍汇编语言源程序书写格式、数据表达式、常用伪指令和正常结束方式。

第 6 章 8086/8088 常用指令：主要介绍数据传送类指令、算术运算类指令、位操作类指令和处理器控制类指令，控制转移类指令在第 7 章节中介绍，字符串处理类指令在第 10 章中介绍。

第 7 章基本程序设计：在介绍控制转移类指令的同时，系统介绍用汇编语言编写程序的步骤和方法，其中包括解决基本问题的算法。

第 8 章子程序设计：介绍子程序的定义、子程序的调用和返回、参数传递，另外，还有子程序的嵌套与递归调用。

第 9 章输入/输出与中断程序设计：在介绍输入和输出的基本概念和四种数据传送方式的基础上，介绍中断系统程序设计。然后介绍 BIOS 中断和 DOS 中断调用。

第 10 章简单应用程序设计：介绍字符串处理类指令及其程序设计，介绍十进制数算术运算调整指令及其程序设计，最后提供应用程序设计举例。

通过教程学习，结合实例上机练习，一般能够基本掌握汇编语言程序设计。

本书由南京师范大学郑阿奇主编，吕升旭编著，参加本书编写的还有梁敬东、顾韵华、王洪元、刘启芬、丁有和、曹弋、徐文胜、殷红先、张为民、姜乃松、彭作民、王一莉、徐斌、王志瑞、周怡明、刘博宇、周怡君、郑进、刘毅等。

本书配有教学课件和所有实例源代码文件，有需要者可以在出版社网站免费下载。网站地址为：http://www.hxedu.com.cn。

由于我们的水平有限，错误在所难免，敬请广大师生、广大读者批评指正。

意见建议邮箱：easybooks@163.com。

编　者

2010 年 11 月

目　录

第 1 章　汇编语言运算基础 ····· 1

1.1　二进制数 ····· 1

1.1.1　二进制 ····· 1

1.1.2　二进制数运算 ····· 3

1.1.3　十六进制数值运算规则 ····· 5

1.1.4　二、八、十六、十进制数相互转换 ····· 6

1.2　计算机中二进制数的表示及运算 ····· 8

1.2.1　不带符号的整数表示 ····· 9

1.2.2　带符号的整数表示 ····· 9

1.2.3　浮点数表示 ····· 12

1.3　计算机中字符表示 ····· 13

1.3.1　ASCII 码 ····· 14

1.3.2　汉字编码 ····· 15

1.4　计算机中逻辑值表示及运算 ····· 17

1.4.1　“与”运算 ····· 17

1.4.2　“或”运算 ····· 17

1.4.3　“非”运算 ····· 18

1.4.4　“异或”运算 ····· 18

1.4.5　逻辑代数基本公式 ····· 19

习题 ····· 20

第 2 章　微型计算机的体系结构 ····· 21

2.1　微型计算机基本组成 ····· 21

2.2　微处理器 ····· 22

2.2.1　Intel 公司微处理器简介 ····· 22

2.2.2　Intel 8086/8088 微处理器功能结构 ····· 23

2.3　Intel 8086/8088 CPU 寄存器组 ····· 26

2.3.1　通用寄存器 ····· 27

2.3.2　段寄存器 ····· 28

2.3.3　控制寄存器 ····· 29

2.4　存储器 ····· 32

2.4.1　存储器的段结构 ····· 32

2.4.2　物理地址与逻辑地址 ····· 33

2.4.3　内存地址空间 ····· 35

2.5　堆栈 ····· 36

2.5.1 堆栈的概念 ····· 36
2.5.2 堆栈的组织 ····· 37
2.5.3 堆栈的操作 ····· 38
2.6 输入/输出设备 ····· 40
习题 ····· 40
第 3 章 汇编语言程序及其操作过程 ····· 42
3.1 机器语言程序 ····· 42
3.1.1 机器语言程序 ····· 42
3.1.2 机器语言程序的输入和运行 ····· 43
3.1.3 Debug 调试工具 ····· 44
3.2 汇编语言程序 ····· 50
3.2.1 汇编语言 ····· 50
3.2.2 汇编语言程序 ····· 50
3.3.3 Emu8086 的使用 ····· 52
3.3 汇编语言程序的编辑、编译、连接和运行 ····· 55
3.3.1 汇编语言程序的编辑 ····· 55
3.3.2 MASM 汇编开发环境 ····· 57
3.3.3 汇编语言程序的编译、连接和运行 ····· 61
习题 ····· 65
第 4 章 汇编语言指令系统和寻址方式 ····· 67
4.1 汇编语言指令的格式 ····· 67
4.2 指令系统分类 ····· 68
4.2.1 数据传送类指令 ····· 68
4.2.2 算术运算类指令 ····· 69
4.2.3 位操作类指令 ····· 70
4.2.4 控制转移类指令 ····· 70
4.2.5 字符串处理类指令 ····· 71
4.2.6 处理器控制类指令 ····· 71
4.3 操作数的类型 ····· 72
4.3.1 立即数操作数 ····· 72
4.3.2 寄存器操作数 ····· 73
4.3.3 存储器操作数 ····· 74
4.4 立即寻址方式 ····· 74
4.5 寄存器寻址方式 ····· 75
4.6 存储器寻址方式 ····· 76
4.6.1 直接寻址方式 ····· 77
4.6.2 寄存器间接寻址方式 ····· 79
4.6.3 寄存器相对寻址方式 ····· 81

4.6.4 基址变址寻址方式 83
4.6.5 相对基址变址寻址方式 85
4.6.6 字符串操作寻址方式 87
习题 87
第 5 章 汇编语言源程序组成 89
5.1 汇编语言源程序的格式 89
5.1.1 段的定义 89
5.1.2 简化段定义格式 93
5.1.3 完整段定义格式 94
5.2 汇编语言数据 96
5.2.1 常量 96
5.2.2 变量 98
5.2.3 标号 103
5.2.4 变量和标号的属性 103
5.3 表达式与运算符 107
5.3.1 算术运算符 107
5.3.2 移位运算符 109
5.3.3 逻辑运算符 109
5.3.4 关系运算符 110
5.3.5 其他运算符 110
5.3.6 运算符的优先级 111
5.4 常用伪指令 112
5.4.1 属性定义伪指令 LABEL 112
5.4.2 段寻址伪指令 ASSUME 113
5.4.3 过程定义伪指令 PROC/ENDP 114
5.4.4 调整偏移量伪指令 115
5.4.5 标题伪指令 TITLE 117
5.4.6 程序结束伪指令 END 118
5.4.7 模块连接伪指令 PUBLIC 和 EXTRN 119
5.5 汇编语言源程序正常结束方式 121
5.5.1 使用 4CH 系统功能调用正常结束 121
5.5.2 使用程序段前缀正常结束 121
习题 123
第 6 章 8086/8088 常用指令 125
6.1 数据传送类指令 125
6.1.1 通用数据传送指令 126
6.1.2 堆栈操作指令 128
6.1.3 数据交换指令 130

6.1.4 查表转换指令 …… 131
6.1.5 地址传送指令 …… 131
6.1.6 标志位传送指令 …… 133
6.1.7 端口输入/输出指令 …… 134
6.2 算术运算类指令 …… 136
6.2.1 加法指令 …… 136
6.2.2 减法指令 …… 139
6.2.3 乘法指令 …… 142
6.2.4 除法指令 …… 144
6.2.5 符号扩展指令 …… 145
6.2.6 BCD 码十进制调整指令 …… 146
6.3 位操作类指令 …… 147
6.3.1 逻辑运算指令 …… 147
6.3.2 移位指令 …… 150
6.3.3 循环移位指令 …… 153
6.4 处理器控制类指令 …… 156
6.4.1 标志位操作指令 …… 156
6.4.2 CPU 控制指令 …… 157
习题 …… 158
第 7 章 基本程序设计 …… 160
7.1 程序设计方法 …… 160
7.1.1 程序设计步骤 …… 160
7.1.2 程序流程图 …… 161
7.2 顺序结构程序设计 …… 162
7.2.1 顺序程序设计 …… 162
7.2.2 简单查表法代码转换 …… 165
7.3 分支程序设计 …… 167
7.3.1 无条件转移指令 …… 168
7.3.2 条件转移指令 …… 172
7.3.3 单分支程序设计 …… 176
7.3.4 双分支程序设计 …… 177
7.3.5 多分支程序设计 …… 179
7.4 循环结构程序设计 …… 186
7.4.1 循环控制指令 …… 186
7.4.2 循环程序的结构 …… 188
7.4.3 单重循环程序设计 …… 189
7.4.4 多重循环程序设计 …… 195
习题 …… 197

第 8 章　子程序设计 …… 199
8.1　子程序的定义 …… 199
8.1.1　子程序的格式 …… 199
8.1.2　子程序的说明信息 …… 200
8.2　子程序的调用和返回 …… 201
8.2.1　调用指令 …… 202
8.2.2　返回指令 …… 205
8.2.3　现场保护和恢复 …… 207
8.3　子程序与主程序间的参数传递 …… 208
8.3.1　利用寄存器传递参数 …… 208
8.3.2　利用堆栈传递参数 …… 210
8.3.3　利用约定存储单元传递参数 …… 213
8.4　子程序的嵌套与递归调用 …… 215
8.4.1　子程序的嵌套调用 …… 215
8.4.2　子程序的递归调用 …… 216
习题 …… 218
第 9 章　输入/输出与中断程序设计 …… 220
9.1　输入和输出的基本概念 …… 220
9.1.1　I/O 端口地址 …… 220
9.1.2　I/O 端口编址 …… 221
9.1.3　I/O 指令 …… 223
9.2　数据传送方式 …… 224
9.2.1　程序控制方式 …… 225
9.2.2　中断控制方式 …… 229
9.2.3　直接存储器存取方式 …… 230
9.3　中断 …… 230
9.3.1　中断的分类 …… 231
9.3.2　中断向量表 …… 232
9.3.3　中断优先级和中断嵌套 …… 235
9.3.4　中断指令 …… 237
9.3.5　中断过程 …… 238
9.3.6　中断处理程序的设计 …… 241
9.4　BIOS 与 DOS 中断调用 …… 243
9.4.1　BIOS 与 DOS 中断概述 …… 243
9.4.2　键盘 BIOS 功能调用 …… 246
9.4.3　键盘 DOS 功能调用 …… 249
9.4.4　显示器 BIOS 功能调用 …… 252
9.4.5　显示器 DOS 功能调用 …… 260

9.4.6 打印机 BIOS 功能调用 …… 261
9.4.7 时钟 BIOS 和 DOS 功能调用 …… 261
习题 …… 262
第 10 章 简单应用程序设计 …… 263
10.1 汇编语言的字符串 …… 263
10.1.1 字符串及其操作命令 …… 263
10.1.2 重复前缀 …… 270
10.2 十进制数算术运算调整指令及其应用 …… 274
10.2.1 压缩 BCD 码的算术运算调整指令 …… 276
10.2.2 非压缩 BCD 码的算术运算调整指令 …… 279
10.3 代码转换 …… 283
10.3.1 其他进制数到二进制数的转换 …… 283
10.3.2 二进制数到其他进制数的转换 …… 291
习题 …… 296
附录 A ASCII 码字符表 …… 298
附录 B DOS 系统功能调用（INT 21H） …… 300
附录 C BIOS 中断调用（INT n） …… 307
附录 D 指令表 …… 312
附录 E 伪指令表 …… 318

第 1 章

汇编语言运算基础

人们最常用的数制是十进制，而计算机中采用二进制。所以，用计算机解决问题时，必须把十进制转换为二进制才能进行运算，运算的结果又要转换为十进制人们才容易阅读。本章介绍二进制的运算及其相关内容。

1.1 二 进 制 数

1.1.1 二进制

为了理解二进制，先从人们熟悉的十进制开始。

1. 十进制

十进制（Decimal）是最常用的数制，其基本特征如下：

（1）基数为 10，采用 0、1、2、3、4、5、6、7、8、9 共十个数码；

（2）逢十进一；

（3）处于不同位置上的数码位权不同。个位上的“1”位权是 1（即 10^0），十位上的“1”位权是 10（即 10^1），百位上的“1”位权是 100（即 10^2），……十进制数整数部分第 n 位的数码位权是 10^{n-1}，从小数点往后数，第 m 位数码位权是 10^{-m}。

例如，十进制数 108.625 用位权表示如下：

$$108.625=1\times10^2+0\times10^1+8\times10^0+6\times10^{-1}+2\times10^{-2}+5\times10^{-3}$$

十进制数 108.625，也可表示为$(108.625)_{10}$或者 108.625D。

2. 二进制

计算机中使用二进制（Binary），其基本特征如下：

（1）基数为 2，采用 0、1 共两个数码；

（2）逢二进一；

（3）二进制数中处于不同位置上的数码代表不同的位权。二进制数整数部分第 n 位的数码位权是 2^{n-1}，从小数点往后数，第 m 位数码位权是 2^{-m}。

例如，二进制数 1101100.101 用位权表示如下：

$$1101100.101=1\times2^6+1\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+0\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}$$

对于二进制小数，小数点向右移一位，数就扩大 2 倍，相当于乘 2；反之，小数点向左移一位，数就缩小 2 倍，相当于除 2。每一位二进制位权对应的十进制如表 1.1 所示。

表 1.1　二进制位权对应的十进制

二 进 制	十 进 制	二 进 制	十 进 制	二 进 制	十 进 制
1	1	10000	16	0.1	0.5
10	2	100000	32	0.01	0.25
100	4	1000000	64	0.001	0.125
1000	8	10000000	128	0.0001	0.0625

二进制数 1101100.101，可以表示为$(1101100.101)_2$，也可表示为 1101100.101B 或 1101100.101b。

3. 八进制

由于 $2^3=8$，一位八进制（Octonary）可以直接对应 3 位二进制。有时用八进制表达相应的二进制。八进制基本特征如下：

（1）基数为 8，采用 0、1、3、4、5、6、7 共八个数码；

（2）逢八进一，借一当八；

（3）八进制数整数部分第 n 位的数码位权是 8^{n-1}，从小数点往后数，第 m 位数码位权是 8^{-m}。

例如，八进制数 154.5 用位权表示如下：

$$(154.5)_8 = 1\times8^2 + 5\times8^1 + 4\times8^0 + 5\times8^{-1}$$

八进制数 154.5 ，可以表示为$(154.5)_8$，也可表示为 154.5Q 或 154.5q。有些书八进制后缀采用字母“O”表示，但字母“O”与数字“0”很像，容易混淆。

4. 十六进制

由于 $2^4=16$，一位十六进制（Hex）可以直接对应 4 位二进制，所以经常用十六进制表达相应的二进制。十六进制基本特征如下：

（1）基数为 16，采用 0、1、3、4、5、6、7、8、9、A、B、C、D、E、F 共十六个数码。其中，A~F(或者 a~f)分别代表十六进制的 10、11、12、13、14、15；

（2）逢十六进一，借一当十六；

（3）十六进制数整数部分第 n 位的数码位权是 16^{n-1}，从小数点往后数，第 m 位数码位权是 16^{-m}。

例如，十六进制数 6C.A 用位权表示如下：

$$(6C.A)_{16} = 6\times16^1 + 12\times16^0 + 10\times16^{-1}。$$

十六进制数 6C.A ，可以表示为$(6C.A)_{16}$，也可表示为 6C.AH 或 6C.Ah。当十六进制数的第 1 位为 A~F(或者 a~f)时，在其前面加上数字 0，例如 0A6H。

表 1.2 列出了 0~15 四种数制之间的对应关系。

也就是说：

$$(14)_{10}=(1110)_2=(16)_8=(E)_{16}$$

或者

$$14D=1110B=16Q=EH$$

表 1.2　常见数制的对应关系

十进制(D)	二进制(B)	八进制(Q)	十六进制(H)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

1.1.2　二进制数运算

二进制数运算主要包括加（+）、减（−）、乘（×）和除（÷）。下面介绍二进制数运算规则，在此基础上说明二进制的优点。

1. 二进制数的运算规则

二进制数比较简单，所以它的运算规则也很简单。

（1）二进制数的加法

根据“逢二进一”规则，二进制数加法的法则如下：

0+0=0　　0+1=1　　1+0=1　　1+1=10

【例 1.1】　求$(1101)_2+(1001.01)_2$

$$\begin{array}{r} 1101 \\ +\ 1001.01 \\ \hline 10110.01 \end{array}$$

所以，$(1101)_2+(1001.01)_2=(10110.01)_2$

（2）二进制数的减法

根据“借一当二”的规则，二进制数减法的法则如下：

0−0=0　　0−1=1(借 1)　　1−0=1　　1−1=0

【例 1.2】　求$(1101)_2-(1001.01)_2$

$$\begin{array}{r} 1101 \\ -\ 1101.01 \\ \hline 0011.11 \end{array}$$

所以，$(1101)_2-(1001.01)_2=(11.11)_2$

（3）二进制数的乘法

二进制数乘法的法则如下：

0×0=0　　0×1=0　　1×0=0　　1×1=1

二进制数的乘法由低位到高位，用乘数的每一位去乘被乘数，若乘数的某一位为 1，则该次部分积为被乘数；若乘数的某一位为 0，则该次部分积为 0。某次部分积的最低位必须和本位乘数对齐，所有部分积相加的结果则为相乘得到的乘积。

【例 1.3】 求$(1001)_2\times(1010)_2$

```
    1001
  × 1010
  ------
    0000
   1001
  0000
 1001
 -------
 1011010
```

所以，$(1001)_2\times(1010)_2=(1011010)_2$

（4）二进制数的除法

二进制数除法的法则如下：

0÷1=0　　1÷1=1

二进制数除法可先从被除数的最高位开始，将被除数（或中间余数）与除数相比较，若被除数（或中间余数）大于除数，则用被除数（或中间余数）减去除数，商为 1，并得相减之后的中间余数；否则商为 0。再将被除数的下一位移下补充到中间余数的末位，重复以上过程，就可得到所要求的各位商数和最终的余数。

注意，除数不能为 0。

【例 1.4】 求$(1110101)_2\div(1001)_2$

```
        1101
1001√1010101
    1001
    ----
     1011
     1001
     ----
       1001
       1001
       ----
          0
```

所以，$(1110101)_2\div(1001)_2=(1101)_2$

2. 二进制的优点

通过对上面二进制的介绍，可以看出二进制具有下列优点：

（1）十进制有十个状态，在自然界中，用某种器件表示十种状态比较难。二进制只有 0 和 1 两个状态，易于实现；

（2）二进制运算规则简单，运算功能比较容易实现。

1.1.3　十六进制数值运算规则

十六进制数的运算可以采用先把该十六进制数转换为十进制数，经过计算后再把结果转换为十六进制数的方法，但这样做比较烦琐。其实，只要按照“逢十六进一”（借一当十六）的规则，直接用十六进制数计算也是很方便的。

1. 十六进制数的加法

十六进制数的加法就是让十六进制数的每一位对应相加，如果和大于或等于 16 时，则进位。

【例 1.5】　求$(05C3)_{16}+(3D25)_{16}$

$$\begin{array}{r} 05C3 \\ +\ 3D25 \\ \hline 42E8 \end{array}$$

所以，$(05C3)_{16}+(3D25)_{16}=(42E8)_{16}$

2. 十六进制数的减法

十六进制数的减法也与十进制数类似，够减时可直接相减，不够减时从向高位借一当十六。

【例 1.6】　求$(3D25)_{16}-(05C3)_{16}$

$$\begin{array}{r} 3D25 \\ -\ 05C3 \\ \hline 3762 \end{array}$$

所以，$(3D25)_{16}-(05C3)_{16}=(3762)_{16}$

3. 十六进制数的乘法

十六进制数的乘法用乘法规则计算，但结果必须用十六进制数表示。

十六进制数的乘法也是由低位到高位，用乘数的每一位去乘被乘数，某次部分积的最低位必须和本位乘数对齐，所有部分积相加的结果则为相乘得到的乘积。

【例 1.7】　求$(3F)_{16}\times(1B)_{16}$

$$\begin{array}{r} 3F \\ \times\ 1B \\ \hline 2B5 \\ 3F \\ \hline 6A5 \end{array}$$

所以，$(3F)_{16}\times(1B)_{16}=(6A5)_{16}$

4. 十六进制数的除法

十六进制数的除法可以仿照十进制数除法规则处理，仅仅需要注意它是十六进制数。在不习惯的情况下，每一步可以先把它变成十进制，计算后的商再变成十六进制商的一个位。

【例 1.8】　求$(6A5)_{16}\div(1B)_{16}$

```
        3F
1B ) 6A5
     51
     195
     195
       0
```

所以，$(65A)_{16}\div(1B)_{16}=(3F)_{16}$

1.1.4 二、八、十六、十进制数相互转换

前面介绍了十进制、二进制、八进制和十六进制，就它们之间数的互相转换方法来说，主要就是十进制数与非十进制数（二、八、十六进制数）互相转换，十进制数与非十进制数之间转换方法类似。二、八和十六进制数之间的转换比较容易。

1. 十进制数与二、八、十六进制数相互转换

十进制数与非十进制数（二、八、十六进制数）转换，包括非十进制数转换为十进制数和十进制数转换为非十进制数。

（1）非十进制数转换为十进制数

根据前面的介绍，非十进制转换为十进制方法为：将非十进制数按权展开，即用位权表示法展开，而后进行相加。

【例 1.9】 将二进制数 101011 转换为十进制数。

$(101011)_2=2^5+2^3+2^1+2^0=32+8+2+1=43$

所以，$(101011)_2=43$

（2）十进制数转换为非十进制数

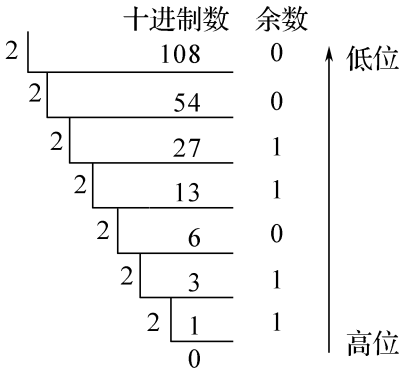
十进制数转换为非十进制数需要对整数部分和小数部分分别进行转换。

整数部分转换方法为“除基取余”法，直到商为 0 为止，将余数倒排即可。

小数部分转换方法为“乘基取整”法，直到乘积的小数部分为 0，或小数点后的位数达到了所需的精度为止，将积的整数部分顺排即可。

【例 1.10】 将十进制数 108.625 转换为二进制数。

整数部分：



小数部分：

```
十进制数  积的整数部分
  0.625
×     2
-------
  1.250      1    高位
  0.25
×     2
-------
  0.5        0
×     2
-------
  1.0        1    低位
```

所以，$(108.625)_{10} = (1101100.101)_2$

【例 1.11】　将十进制数 108.625 转换为八进制数。

整数部分：

```
十进制数    余数
8 | 108      4    低位
  8 | 13     5
    8 | 1    1    高位
      0
```

小数部分：

```
十进制数  积的整数部分
  0.625
×     8
-------
  5.00       5
```

所以，$(108.625)_{10} = (154.5)_8$

【例 1.12】　将十进制数 108.625 转换为十六进制数。

整数部分：

```
十进制数    余数
16 | 108     12    低位
   16 | 6     6    高位
      0
```

小数部分：

```
十进制数  积的整数部分
  0.625
×    16
-------
  10.0       10
```

所以，$(108.625)_{10} = (6C.A)_{16}$

通过例 1.10、例 1.11 和例 1.12 可知：$(108.625)_{10} = (1101100.101)_2 = (154.5)_8 = (6C.A)_{16}$

或者，108.625=1101100B=154.5Q=6C.AH

2. 二进制数与八、十六进制数相互转换

因为 $2^3=8$，$2^4=16$，所以二进制数可以直接对应它的八进制数和十六进制数。同理，八进制数和十六进制数也可以直接对应它的二进制数。

（1）二进制数转换为八、十六进制数

二进制数转换成八进制数的方法为：

以小数点为基准，整数部分从右向左，每 3 位为一组，最左边不足 3 位时，左边添 0 补足 3 位；小数部分从左向右，每 3 位为一组，最右边不足 3 位时，右边添 0 补足 3 位。然后将每组中的 3 位二进制数用一位八进制表示，依序排列即可得到对应的八进制数。

【例 1.13】 将二进制数 10101001100.1101 转换为八进制数。

10101001100.1101 = 010 101 001 100.110 100

= 2 5 1 4 . 6 4

所以，$(10101001100.1101)_2=(2514.64)_8$

二进制数转换成十六进制数的方法为：

以小数点为基准，整数部分从右向左，每 4 位为一组，最左边不足 4 位时，左边添 0 补足 4 位；小数部分从左向右，每 4 位为一组，最右边不足 4 位时，右边添 0 补足 4 位。然后将每组中的 4 位二进制数用一位十六进制表示，依序排列即可得到对应的十六进制数。

【例 1.14】 将二进制数 10101001100.1101 转换为十六进制数。

10101001100.1101 = 0101 0100 1100.1101

= 5 4 C . D

所以，$(10101001100.1101)_2=(54C.D)_{16}$

（2）八、十六进制数转换为二进制数

八进制数转换成二进制数的方法为：

将每位数码用 3 位二进制数码替换，而后依序排列即可。整数部分最前面的 0 和小数部分最后面的 0 可以省略。

十六进制数转换成二进制数的方法为：

将每位数码用 4 位二进制数码替换，而后依序排列即可。整数部分最前面的 0 和小数部分最后面的 0 可以省略。

【例 1.15】 将八进制数 154.42 转换为二进制数，将十六进制数 D31. 2C 转换为二进制数。

$(154.42)_8 = (001\ 101\ 100.100\ 010)_2$

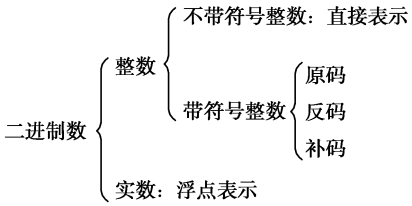
$= (1101100.100\ 01)_2$

$(D31.2C)_{16} = (1101\ 0011\ 0001.0010\ 1100)_2$

$= (110100110001.001011)_2$

1.2 计算机中二进制数的表示及运算

二进制数分为整数和实数两类，而整数又分为不带符号的整数和带符号的整数。整数和实数在计算机中表示如下：



1.2.1　不带符号的整数表示

二进制数表示的不带符号整数的最小数值是全 0，最大数值是全 1。例如，8 位二进制数表示的不带符号整数的范围是 00000000～11111111，对应十进制范围为 0～255。

就像 $999=10^3-1$，$(11111111)_2=2^8-1=255$。

同样，16 位二进制数表示的不带符号整数的范围是 0000000000000000～1111111111111111，对应十进制范围为 $0\sim2^{16}-1$，即 0～65535。

n 位二进制数表示的不带符号整数的范围是 $0\sim2^n-1$。

计算机存放数据是规定位数的，不满规定位数时，不带符号整数前面用 0 填充。

例如，108 在计算机中 8 位表示的数据如下：

0	1	1	0	1	1	0	0

108 在计算机中 16 位表示的数据如下：

0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0

在计算机中，除了表示不带符号的整数，还需要表示带符号的整数及实数。

1.2.2　带符号的整数表示

在数学中，将正号“+”和负号“-”放在数字的前面表示数值的正负。为了在计算机中表示数值的符号，人们将数值的符号“数码化”：用“0”表示正号，用“1”表示负号。

符号被“数码化”的数称为机器数，而符号没有被“数码化”的数称为数的真值。机器数一般是固定长度的，这个长度等于计算机的字长，即计算机一次能同时进行二进制整数运算的位数。机器数长度一般是 8 的倍数，如 8 位、16 位、32 位和 64 位等。“数码化”后的符号位放在整个字长位数的最高位。

在计算机中，带符号的整数用原码或补码表示。

1. 原码

原码是一种简单的机器数表示法，它规定机器数中的最高位（最左边的位）是符号位，“0”代表正号，“1”代表负号，以后各位是该数的绝对值。

例如，8 位二进制原码表示如图 1.1 所示。

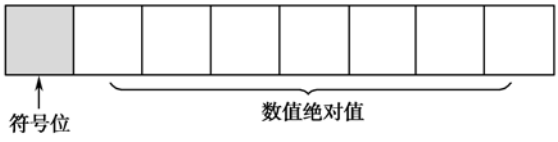


图 1.1　二进制原码表示

8 位原码表示的数值范围是 **1**1111111～**0**1111111，即−(2^7−1)～+(2^7−1)，对应的十进制数为−127～+127。

16 位原码最高位是符号位，以后各位是该数的绝对值。16 位原码表示的数值范围是− (2^{15}−1)～+(2^{15}−1)，对应十进制数为−32767～+32767。

n 位原码表示的数值范围是− (2^{n-1}−1)～+(2^{n-1}−1)，若要表达的数超过该范围时，将产生溢出。

【例 1.16】 已知两个十进制数 a=108，b=−43，写出在计算机中 8 位和 16 位的原码表示。

因为 108=$(1101100)_2$，43=$(101011)_2$，则：

十进制数	真值	8 位原码	16 位原码
108	+1101100	**0**1101100	**0**000000001101100
43	+ 101011	**0**0101011	**0**000000000101011
−43	−101011	**1**0101011	**1**000000000101011

A=108，b=−43，a、b 的原码写成：$[a]_{原}$=01101100 ，$[b]_{原}$=10101011。

机器数用原码表示简单易懂，但进行加减运算比较复杂。因为原码把符号“数码化”了，其运算方法与手算类似。例如，要做 a＋b 的运算，首先要判别符号，若 a、b 同号，则相加；若 a、b 异号，则要判断两数绝对值的大小，然后将绝对值大的数减去绝对值小的数，结果符号取绝对值大的数的符号。显然，这种运算方法不仅增加运算时间，而且使运算部件复杂了。

2. 补码

补码是另一种机器数表示法，它的指导思想是：不管两个数相加还是相减，只要变成两个数的补码相加即可。而且，不再需要单独处理符号位，符号位与数一起参加运算。例如，有两个二进制数 a、b，可以将 a、b、−a、−b 的补码写成$[a]_{补}$、$[b]_{补}$、$[-a]_{补}$、$[-b]_{补}$，计算 a+b 变成计算$[a]_{补}$+$[b]_{补}$；计算 a−b 就可变成计算$[a]_{补}$+$[-b]_{补}$。

对于正数而言，其补码与原码相同；对于负数而言，其补码最高位的符号位为 1，其余对原码中相应位取反后，再加 1。

为了计算补码，引入了反码。对于正数而言，其反码与原码相同；对于负数而言，其反码除了符号位为 1，其余对原码中相应位取反。这样，负数的补码=负数的反码+1。

【例 1.17】 已知两个十进制数 a=108，b=−43，计算它们在计算机中 8 位和 16 位的补码表示。

在计算机中 8 位和 16 位的原码、反码、补码如下：

108原码：	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0
反码：	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0
补码：	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0
−43原码：	1	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1
反码：	1	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0
补码：	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1

与原码相对应，8 位补码表示的数值范围是：

原码：**1**1111111～**0**1111111，即−(2^7−1)～+(2^7-1)，−127～+127

补码：**1**0000001～**0**1111111

实际上，补码 10000000 表示−128，这样 8 位补码表示的数值范围是−128～+127，即−(2^7)～+(2^7−1)。*n* 位原码表示的数值范围是−(2^{n-1})～+(2^{n-1}−1)，要表达的数超过该范围时，将产生溢出。

3. 原码补码真值对照

n 位补码求真值的方法是：当机器数的最高位（符号位）为 0 时，表示真值是正数，其值等于其余 *n*−1 位的值；当机器数的最高位（符号位）为 1 时，表示真值是负数，其值等于其余 *n*−1 位按位取反末位加 1 的值。

例如：

若$[a]_{补}$=**0**1111111，则 $a=(+1111111)_2=(+127)_{10}$

若$[a]_{补}$=**1**1111111，则 $a=(-0000001)_2=(-1)_{10}$

表 1.3 给出了 8 位二进制码对应的无符号二进制数、原码及补码的值。

表 1.3　计算机中的 8 位数据表示的值对比

二进制数码表示	无符号二进制数	原　　码	补　　码
00000000	0	+ 0	+ 0
00000001	1	+ 1	+ 1
00000010	2	+ 2	+ 2
⋮	⋮	⋮	⋮
01111110	126	+ 126	+ 126
01111111	127	+ 127	+ 127
10000000	128	−0	−128
10000001	129	−1	−127
10000010	130	−2	−126
⋮	⋮	⋮	⋮
11111110	254	−126	−2
11111111	255	−127	−1

4. 二进制补码运算

负数用补码表示后，就可以和正数一样处理。二进制数补码加减运算规则如下：

$$[a+b]_{补}=[a]_{补}+[b]_{补}$$

$$[a-b]_{补}=[a]_{补}+[-b]_{补}$$

由运算规则可知，在用补码做加法运算时，两个补码直接相加的结果就是和的补码。而用补码做减法运算时，可以将减法运算变为加法运算，即先将减一个数变成加上这个数的负数，然后再补码求和。

当带符号的两个数采用补码形式表示时，加法运算把符号位和数值位一起进行运算（若符号位有进位，则丢弃），结果为两数之和的补码形式。所以，运算器只需要一个加法器就可以了，不必再配一个减法器，而且也不必对符号位进行专门处理。

【例 1.18】　已知两个十进制数 a=108，b=43 存储在字长为 8 位的计算机中，请利用补码运算，求 a+b、a−b、−a+b 的结果。

先分别求出 a、b、−a、−b 的补码：

$[a]_{补}$=01101100，$[b]_{补}$=00101011，$[-a]_{补}$=10010100，$[-b]_{补}$=11010101

（1）计算 a+b

十进制	二进制
108	**0**1101100
+ 43	+ **0**0101011
151	**1**0010111（溢出）

因为 8 位的计算机中，最高位表示符号，所以，最大表达的数据只能是 7 个 1，即 127，而实际数据 151 已经超过了表达的范围，造成结果不正确。

如果参加运算的数据的符号位相同，而运算的结果的符号位不同，如下所示：

0xxxxxxx	1xxxxxxx
+ 0xxxxxxx	+ 1xxxxxxx
1xxxxxxx	0xxxxxxx

运算结果产生溢出。计算机通过判断两个相同的符号与运算结果符号不同来设置运算结果的溢出标志。

（2）计算 a−b

十进制	二进制	
108	**0**1101100	
− 43	+ **1**1010101	
65	**1**01000001	（符号位的进位将丢弃）

因为 8 位的计算机中，只能存放 8 位数据，最高位运算产生的进位只能存放到运算结果的进位标志位中，跟运算结果无关。运算后，最高位仍然是符号位。

因此，a−b= $(1000001)_2$=65

（3）计算−a+b

十进制	二进制
−108	**1**0010100
+ 43	+ **0**0101011
−65	**1**0111111
	11000001（原码）

如果运算结果为负数（符号位为 1），则将补码变成原码才可以看出结果。

因此，−a+b= $(-1000001)_2$=−65

1.2.3 浮点数表示

上面介绍了计算机中整数的表示，而实数可以是小数，也可以既包含整数又包含小数。例如：

十进制实数：123.456；−0.001234；

二进制实数：1011.011；−0.000101101；

在计算机中，二进制实数可以用浮点表示。为了理解浮点的表示方法，先看一下科学计数法。在科学计数法中，任意一个十进制实数可以表示成一个纯小数和一个 10 的指数

乘幂的积。例如：

$$123.456 = 0.123456\times10^{3}，\ -0.00123456 = (-0.123456)\times10^{-2}$$

同样，二进制实数也可以表示成类似的形式，例如：

$$1011.011=0.1011011\times2^{100}$$

$$-0.000101101=(-0.101101)\times2^{-11}$$

为了表达二进制数，可以仅需要一个二进制纯小数和一个二进制的指数。在计算机中，为了保证表示方式的唯一性，根据 IEEE754 国际标准，对实数需要进行规格化处理。规格化后的小数形式为 1.M，这时的小数部分(M)称为尾数。而指数部分称为阶码，IEEE754 规定，阶码用指数加上一个数后形成的移码表示。这就是计算机浮点数表示。

例如：

$$1011.011=1.011011\times2^{11}$$

$$-0.000101101=(-1.01101)\times2^{-100}$$

浮点数有两种格式，单精度浮点数采用 32 位表示，双精度浮点数采用 64 位表示。单精度浮点数计算机内表示如图 1.2 所示。



图 1.2　32 位浮点数的一种表示法

其中，

符号：数值为正，符号位为 0；数值为负，符号位为 1。

阶码：采用移码表示，移码 = 指数+127（即 8 位二进制 01111111）。

尾数：为规格化后的小数部分。

【例 1.19】　采用 IEEE754 单精度浮点数表示二进制实数 1011.011 和−0.001011011。

因为$1011.011=1.011011\times2^{11}$，所以，

移码：11+01111111=10000010

计算机浮点数表示如下：

0	10000010	01101100000000000000000

因为$-0.000101101=(-1.01101)\times2^{-100}$，所以

移码：−100+01111111=01111011

计算机浮点数表示如下：

1	01111011	01101000000000000000000

1.3　计算机中字符表示

在计算机中，二进制除了表示数值、逻辑值外，还可以表示字符信息。

日常生活中经常使用的字符包括西文字符和汉字等。此外，世界上还有许多其他的文字需要表示。为了在计算机中表达这些字符，需要用不同的二进制数的组合表示，这种组合就是二进制编码。根据不同的用途，有各种各样的编码方案。

字符编码
- 西文字符：ASC II码
- 汉字字符：GB2312—80
- 全世界所有字符：Uncode码

1.3.1 ASCII 码

ASCII 码（American Standard Code for Information Interchange）即美国标准信息交换码，是目前全世界计算机中使用得最广泛的西文字符集编码。西文字符集由拉丁字母、数字、标点符号及一些特殊符号组成，字符集中的每一个字符都用一个数表示，称为该字符的编码，用二进制表示的 ASCII 码如表 1.4 所示。

表 1.4 标准 ASCII 码表

高位 / 低位	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

标准 ASCII 码用 7 位二进制数编码，它可以表示 128（即 2^7）个字符，称为 ISO646 标准。表中包括控制字符和可打印字符。

其中，

控制字符：00H～1FH、7FH，共 33 个。

可打印字符：20H～7EH，共 95 个。

1. 控制字符

标准 ASCII 码表中，00H～20H 部分属于控制字符，主要用于通信控制或对计算机设备的控制，例如，NUL（NULL）为空字符，BEL（BELL）为响铃，LF（Line Feed）为换行，FF（Form Feed）为换页，CR（Carriage Return）为回车。

2. 可打印字符

标准 ASCII 码表中，21H～7EH 部分属于可打印字符，表示如下：

字　　符	二进制 ASCII 码	十六进制 ASCII 码	十进制 ASCII 码
0	0110000B	30H	48D
A	1000001B	41H	65D
a	1100001B	61H	97D
空格	0100000B	20H	32D

在计算机内部，以 8 位二进制位（称为一个字节）存放一个字符，ASCII 码仅需要 7 位二进制。每个字节空出的最高位，通常为 0。例如，字符“A”对应的 ASCII 码“1000001”在计算机中表示如下：

0	1	0	0	0	0	0	1

【例 1.20】　写出“Hello World!”所对应的二进制编码。

根据 ASCII 码表，“Hello World!”所对应的二进制编码如下：

01001000　01100101　01101100　01101100　01101111　00100000　01010111　01101111　01110010

H　e　1　1　o　空格　W　o　r

01101100 01100100 00100001

1　d　!

1.3.2　汉字编码

1. 国标码

1981 年，我国颁布信息交换汉字编码的第一个国家标准，称为 GB2312—80。该标准包含的字符集由如下三部分组成：

字母、数字和各种符号，包括拉丁字母、俄文、日文平假名与片假名、希腊字母、汉语拼音等共 682 个；

一级常用汉字，共 3755 个，按汉语拼音排列；

二级常用字，共 3008 个，按偏旁部首排列。

GB2312—80 字符集构成一个二维码表，它分成 94 行和 94 列，行号称为区号，列号称为位号，如图 1.3 所示。

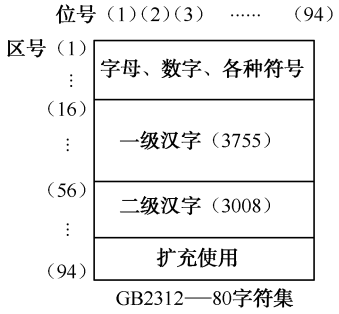


图 1.3　GB2312—80 字符集

每一个汉字或符号在码表中都有各自的位置，即所在的区号及位号。用区号及位号组合在一起的编码就是该汉字区位码，其中 01 区和 54 区字符如图 1.4 所示。

国标第01区

	0	1	2	3	4	5	6	7	8	9
0			、	。	?	ˉ	ˇ	¨	〃	々
1	?	～	‖	…	‘	’	“	”	〔	〕
2	〈	〉	《	》	「	」	『	』	〖	〗
3	【	】	±	×	÷	∶	∧	∨	Σ	Π
4	∪	∩		∷	√	⊥	//	∠	⌒	⊙
5	∫	∮	≡	≌	≈	∽	∝	≠	≮	≯
6	≤	≥	∞	∵	∴	♂	♀	°	′	〃
7	℃	$	¤	¢	£	‰	§	№	☆	★
8	○	●	◎	◇	◆	□	■	△	▲	※
9	→	←	↑	↓	〓					

国标第54区

	0	1	2	3	4	5	6	7	8	9
0		帧	症	郑	证	芝	枝	支	吱	蜘
1	知	肢	脂	汁	之	织	职	直	植	殖
2	执	值	侄	址	指	止	趾	只	旨	纸
3	志	挚	掷	至	致	置	帜	峙	制	智
4	秩	稚	质	炙	痔	滞	治	窒	中	盅
5	忠	钟	衷	终	种	肿	重	仲	众	舟
6	周	州	洲	诌	粥	轴	肘	帚	咒	皱
7	宙	昼	骤	珠	株	蛛	朱	猪	诸	诛
8	逐	竹	烛	煮	柱	瞩	嘱	主	著	柱
9	助	蛀	贮	铸	筑					

图 1.4　GB2312—80 字符集 01 区和 54 区字符

例如，“中”字的区号是 54，位号是 48，区位码是 5448。为了与 ASCII 码中可打印字符位置表达一致，需要加上 20H，从而得到汉字国标码。

即，国标码=区位码+2020H

例如：

“中”区位码是 5448，十六进制表示为 3630H，

“中”国标码=3630H+2020H=5650H，二进制表示为 01010110 01010000

2. 汉字内码

汉字内码是计算机内部对汉字进行存储、处理和传输的汉字代码。由于西文字符的内码就是标准 ASCII 码，用一个字节表示，最高位为 0；而国标码的两个字节中，每个字节的最高位也都是 0，容易引起混淆和冲突。为了保证中西文兼容，就将国标码的每个字节的最高位置为 1，构成汉字机内码，简称内码。即内码的形式如下：

1	国标码第一字节	1	国标码第二字节

即：汉字内码 = 国标码+8080H。

目前 PC 中 GB2312—80 汉字内码的表示都是这种方式。

例如，“中”字的机内码是：

11010110 11010000 = 01010110 01010000+1000000010000000。

【例 1.21】“腾讯 QQ 2011 版。”二进制编码如下：

1100110011011010 1101000110110110 01010001 01010001 00100000 00110010

腾　　讯　　Q　　Q　　空格　　2

00110000 00110001 00110001 1011000011100110 1010000110100011

0　　1　　1　　版　　。

这段编码共占 75 个字节，其中有西文字符（ASCII 码），也有汉字（汉字内码）。计算机通过每一个字节的最高位为 0 和 1 区分它们。

除了 ASCII 码和 GB2312 外，Unicode 码也非常流行。

1.4　计算机中逻辑值表示及运算

计算机不仅可以进行数值运算，例如对二进制数进行加减乘除等各种运算。而且还具有很强的逻辑运算能力，即可以对逻辑值进行运算。

逻辑值只有两个：真或假。在计算机中就使用的二进制数的两个数码来表示，一般用数字“1”代表真，用数字“0”代表假。

常用的基本逻辑运算有“与”、“或”、“非”和“异或”。

1.4.1　“与”运算

如果决定一件事需要两个以上条件，且缺一不可，那么结果与各条件的关系称为与关系。逻辑“与”也叫做逻辑乘，一般用符号“∧”或“·”表示。假设 A 和 B 为逻辑变量，F 表示逻辑运算结果，则逻辑“与”的运算表示为：

$$F=A \wedge B \quad 或 \quad F=A \cdot B$$

逻辑“与”的功能是：仅当逻辑变量 A 与 B 的值均为 1 时，运算结果 F 为 1。也就是当两个逻辑变量取值都为“真”时，结果才为“真”。其他情况下，逻辑“与”运算结果为 0（逻辑“假”）。逻辑“与”的运算规则如表 1.5 所示。

表 1.5　逻辑与运算规则

A	B	F=A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

【例 1.22】　设 x = 0FH，y = 55H，求 f = x ∧ y。

$$\begin{array}{r} x = 0000\,1111\,B \\ \wedge\ y = 0101\,0101\,B \\ \hline f = 0000\,0101\,B \end{array}$$

所以，0FH ∧ 55H = 05H。

1.4.2　“或”运算

如果决定一件事可以有两个以上条件，且只要有一个条件满足时该事就行，那么结果与各条件的关系称为“或”。逻辑“或”也叫做逻辑加，一般用符号“∨”或“+”表示，但使用“+”时不能与表达式中的其他算术运算相混淆。设 A 和 B 为逻辑变量，F 表示逻辑运算结果，则逻辑“或”的运算表示为：

$$F=A \vee B \quad 或 \quad F=A+B$$

逻辑“或”的功能是：当逻辑变量 A 或 B 中至少有一个为 1 时，运算结果 F 为 1，即只要有一个条件为“真”或两个为真，结果就为“真”。仅当两个逻辑变量均为 0 时，运

算结果才为 0。逻辑“或”的运算规则如表 1.6 所示。

表 1.6 逻辑或运算规则

A	B	F=A ∨ B
0	0	0
0	1	1
1	0	1
1	1	1

【例 1.23】 设 x=0FH，y=55H，求 f=x ∨ y。

$$\begin{array}{r} x = 0000\,1111\,B \\ \vee\ y = 0101\,0101\,B \\ \hline f = 01011111\,B \end{array}$$

所以，0FH ∨ 55H =5FH。

1.4.3 “非”运算

如果条件与结果相反，当条件满足时结果不成立；条件不满足时结果成立，那么结果与条件之间的关系称为“非”。逻辑“非”运算通常是在逻辑变量上方加一横线表示，如 $\overline{A}$、$\overline{B}$。

逻辑“非”运算的功能是：当逻辑变量为 1 时，其运算结果为 0；而当逻辑变量为 0 时，运算结果为 1。逻辑“非”的运算规则如表 1.7 所示。

表 1.7 逻辑非运算规则

A	F= $\overline{A}$
0	1
1	0

【例 1.24】 设 x=0FH，求 f= $\overline{x}$。

$$x=0000\ 1111B$$

$$f=\overline{x}=1111\ 0000B$$

所以，$\overline{x}$ =F0H。

1.4.4 “异或”运算

如果决定一件事有两个条件，当只有一个条件满足时就可行，两个条件都满足或两个条件都不满足时不可行，那么结果与各条件的关系称为“异或”关系。“异或”运算通常用符号“∀”或“⊕”表示。设 A 与 B 为逻辑变量，则逻辑“异或”表示为：

$$F=A\forall B \quad 或 \quad F=A\oplus B$$

逻辑“异或”运算功能是：两个逻辑变量 A 和 B 的取值相同，运算结果则为 0；A 与 B 的取值不同（一个为 1，另一个为 0）时，运算结果为 1。这个功能可简记为“相同为 0，不同为 1”。“异或”运算规则如表 1.8 所示。

表 1.8　逻辑异或运算规则

A	B	F=A∀B
0	0	0
0	1	1
1	0	1
1	1	0

【例 1.25】　设 x=0FH，y=55H 求 f=x ⊕ y。

$$\begin{array}{r} x = 0000\,1111\,B \\ \oplus\ y = 0101\,0101\,B \\ \hline f = 01011010\,B \end{array}$$

所以，0FH ⊕ 55H =5AH。

1.4.5　逻辑代数基本公式

为了理解逻辑代数的基本公式，先通过下面例子看十进制算术运算的基本公式。

例如，16×(25+35)=16×25+16×35

有十进制数 A、B、C，则：A×(B+C)= A×B+A×C。这就是一个十进制算术运算的基本公式。十进制算术运算的基本公式很多，例如，A×0=0；A×1=1；A+(B+C)=(A+B)+C；A× (B×C)=(A×B) ×C；等等。

同理，逻辑代数运算也有很多基本公式，如表 1.9 所示。

表 1.9　逻辑代数基本公式

序　号	公　式	序　号	公　式
1	$0\cdot A=0$	10	$\overline{1}=0$；　$\overline{0}=1$
2	$1\cdot A=A$	11	$1+A=1$
3	$A\cdot A=A$	12	$0+A=A$
4	$A\cdot \overline{A}=0$	13	$A+A=A$
5	$A\cdot B=B\cdot A$	14	$A+\overline{A}=1$
6	$A\cdot(B\cdot C)=(A\cdot B)\cdot C$	15	$A+B=B+A$
7	$A\cdot(B+C)=A\cdot B+A\cdot C$	16	$A+(B+C)=(A+B)+C$
8	$\overline{A\cdot B}=\overline{A}+\overline{B}$	17	$A+B\cdot C=(A+B)\cdot(A+C)$
9	$\overline{\overline{A}}=A$	18	$\overline{A+B}=\overline{A}\cdot\overline{B}$

【例 1.26】　设 x=0FH，y=55H，求 $f=\overline{x\cdot y}$。

可以先计算 $x\cdot y$，然后再计算 $\overline{x\cdot y}$。

又因为 $\overline{x\cdot y}=\overline{x}+\overline{y}$，所以也可以先求出 $\overline{x}$ 和 $\overline{y}$：

$\overline{x}$ =1111 0000 B，$\overline{y}$ =1010 1010B，则：$\overline{x}+\overline{y}$ =1111 1010B

所以，$\overline{x\cdot y}$ =FAH。

习题

1．试完成下列数制间的转换。

（1）十进制数转换为二进制数。

65，89，139，257，388，86

（2）无符号二进制数转换为十进制数。

01010010，11010010，00101110，11101100，10011010，00011011

（3）十进制数转换为十六进制数。

23，689，356，456，879，484

（4）八进制数转换为十六进制数。

475，652，752，346，44，12

2．下列 4 种不同进制的数中，其值最大的是_____。

A．$(11011101)_2$　B．$(80)_{10}$　C．$(67)_8$　D．$(3C)_{16}$

3．下列 4 种不同进制的数中，其值最小的是_____。

A．10100100B　B．30D　C．27Q　D．4AH

4．求下列带符号十进制数的 8 位基 2 码（二进制码）补码。

（1）+127　（2）−1　（3）−128　（4）+1

5．求下列带符号十进制数的 16 位基 2 码（二进制码）补码。

（1）+655　（2）−1　（3）−1312　（4）+101

6．已知 8 位二进制数 10111010（最高位为符号位），当它是原码时，表示的十进制数是（1）_____；当它是反码时，表示的十进制数是（2）_____；当它是补码时，表示的十进制数是（3）_____。

（1）A．−69　B．−58　C．−6　D．−5

（2）A．71　B．70　C．−69　D．−70

（3）A．71　B．70　C．−69　D．−70

7．设机器码的长度为 8 位，已知 x、y 为带符号纯整数，z 为带符号纯小数，$[x]_{原}=[y]_{补}=[z]_{移}=11111111$，求出 x、y、z 的十进制真值，x=（1）_____，y=（2）_____，z=（3）_____。

（1）A．1　B．127　C．−127　D．−1

（2）A．1/128　B．−1/128　C．−127/128　D．127/128

（3）A．−1　B．127　C．−127　D．1

8．将十进制实数 123.456 分别转换为二进制数、八进制数和十六进制数。

9．写出下列有符号十进制数的 8 位二进制数补码与 16 位二进制数补码，总结符号扩展的方法。

（1）−50　（2）−122　（3）110　（4）−94

10．写出字符 0、9、A、a、Z、z 的 ASCII 码，比较相邻两个字符间 ASCII 码值之差的大小。

第 2 章

微型计算机的体系结构

硬件是软件的基础，软件是硬件的延伸。计算机程序只有在计算机上才有生命。因此学习计算机语言，特别是汇编语言，必须首先熟悉计算机的硬件和基本工作原理，因为汇编语言是和计算机硬件直接交互的程序设计语言。因此，本章对计算机的体系结构和组成进行了探讨，并且重点介绍了 Intel 8086/8088 微处理器的功能结构、寄存器组成、存储器组织和堆栈设置等方面的知识。

2.1 微型计算机基本组成

微型计算机主要由五大部件组成：运算器、控制器、存储器、输入设备和输出设备，它们之间通过数据总线、地址总线和控制总线连接。通常将运算器和控制器称为中央处理器（Center Process Unit，CPU），对于微型计算机，又称为微处理器。输入设备和输出设备（简称 I/O）通过输入/输出接口电路与 CPU 相连。微型计算机系统的结构如图 2.1 所示。

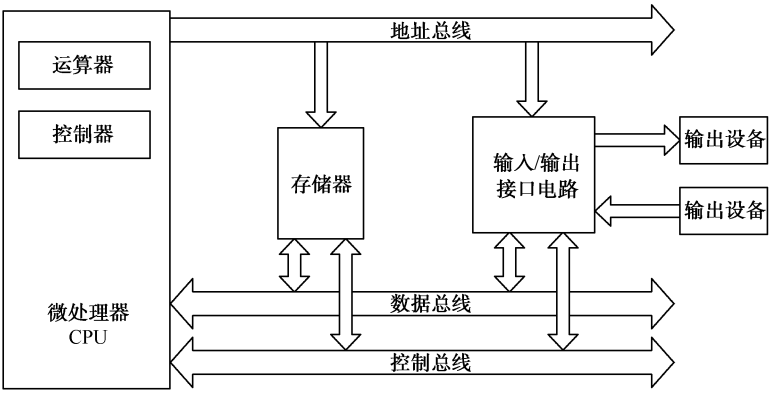


图 2.1　微型计算机结构图

下面介绍计算机的三个总线。

（1）地址总线。地址总线由若干根地址线组成，它的任务是传递 CPU 要访问数据的地址。地址总线的根数（宽度）决定了 CPU 的寻址能力。如果有 20 根地址总线，则表示 CPU 最多可以寻址 1M 个单元。

（2）数据总线。CPU 与存储器和输入/输出接口电路之间数据的传递是通过数据总线进行的。数据总线的根数（宽度）决定了 CPU 和外界一次传递数据的位数。8 根数据总线一次可传递一组 8 位二进制数据。而数据总线宽度为 16 处理器一次可以传递一组 16 位二进制数。

（3）控制总线。控制总线是一个总称，它是由不同的控制线组成的。CPU 通过控制总线对存储器和输入/输出接口电路进行控制，并且得到状态信息。控制总线的宽度越宽，计算机的控制能力就越强。

2.2 微 处 理 器

美国 Intel 公司是目前世界上最有影响的微处理器生产厂家，也是世界上第一个微处理器生产厂家，他们所生产的 80x86 系列微处理器一直是个人计算机的主流 CPU。

2.2.1 Intel 公司微处理器简介

自 20 世纪 70 年代开始出现微型计算机以来，CPU 经历了飞速的发展。1971 年，Intel 设计成功了第一片 4 位微处理器 Intel 4004；随之又设计生产了 8 位微处理器 8008；1973 年推出了 8080；1974 年基于 8080 的个人计算机（Personal Computer，PC）问世，Microsoft 公司的创始人 Bill Gates 为 PC 开发了 BASIC 语言解释程序；1977 年 Intel 推出了 8085。自此之后，Intel 又陆续推出了 8086、80386、Pentium 等 80x86 系列微处理器。各种微处理器的主要区别在于处理速度、寄存器位数、数据总线宽度和地址总线宽度。

1981 年，美国国际商业机器公司（International Business Machine Corporation，IBM）采用 8088 CPU 推出了具有划时代意义的个人计算机，简称 IBM PC。次年年底又推出了展型 PC/XT。1984 年，IBM 采用 80286 推出了增强型 IBM PC/AT。IBM PC/XT/AT 这三款个人计算机通常被称为 PC 系列机，它们都是 16 位的。后来的各种 32 位 PC 都是采用 32 位 Inte l80x86 CPU 而形成的计算机，但其基本结构仍然源于 PC/AT。现在人们谈到的 PC 或微机就是上述微型计算机系统的统称。

下面简要介绍不同时期 Intel 公司制造的几种主要型号的微处理器，这些微处理器都曾经或正在广为流行。

1. 80x86 系列微处理器

（1）8088 微处理器。具有多个 16 位寄存器、8 位数据总线和 20 位地址总线，可以寻址 1MB 的内存。虽然这些寄存器一次可以处理 2 个字节，但数据总线一次只能传送 1 个字节。该处理器只能以实模式工作。

（2）8086 微处理器。指令系统与 8088 完全相同，具有多个 16 位寄存器、16 位数据总线和 20 位地址总线，可以寻址 1MB 的内存，一次可以传送 2 个字节。该处理器只能以实模式工作。

（3）80286 微处理器。比 8086 运行更快，具有多个 16 位寄存器、16 位数据总线和 24 位地址总线，可以寻址 16MB 内存。它既可以以实模式工作，也可以以保护模式工作。

（4）80386 微处理器。具有多个 32 位寄存器、32 位数据总线和 32 位地址总线，可以寻址 4GB 内存。它提供了较高的时钟速度，增加了存储器管理和相应的硬件电路，减少

了软件开销，提高了效率。它既可以以实模式工作，也可以以保护模式工作。

（5）80486 微处理器。具有多个 32 位寄存器、32 位数据总线和 32 位地址总线。它比 80386 增加了数字协处理器和 8KB 的高速缓存，提高了处理速度。它既可以以实模式工作，也可以以保护模式工作。

（6）Pentium（奔腾）。具有多个 32 位寄存器、64 位数据总线和 36 位地址总线。因为它采用了超标量体系结构，所以每个时钟周期允许同时执行 2 条指令，处理速度得到了进一步提高，性能比 80486 优越得多。它既可以以实模式工作，也可以以保护模式工作。

以上介绍了 Intel 80x86 系列的一些主要微处理器，表 2.1 给出了该系列部分微处理器的数据总线和地址总线宽度。实际上 80x86 系列的功能还在不断改进和增强，它们的速度将会更快，性能将会更优越。但无论怎样变化，它们总会被设计成是完全向下兼容的，就像在 8086 上设计和运行的软件，可以不加任何改变地在 Pentium 4 机上运行一样。对于汇编语言编程人员来讲，掌握 16 位计算机的编程十分重要，它是学习高档计算机及保护模式编程的基础，也是掌握实模式程序设计的唯一方法。

表 2.1　Intel 80x86 系列微处理器总线宽度

CPU	数据总线宽度	地址总线宽度	CPU	数据总线宽度	地址总线宽度
8086	16	20	Pentium	64	36
8088	8	20	Pentium Ⅱ	64	36
80286	16	24	Pentium Ⅲ	64	36
80386SX	16	24	Pentium 4	64	36
80386DX	32	32	Itanium	64	44
80486	32	32			

2. CPU 主要性能指标

（1）机器字长。机器字长和 CPU 内部寄存器、运算器、内部数据总线的位宽相一致。如 8086 CPU，它的内部寄存器是 16 位的，运算器能完成 16 位二进制数运算。数据总线的位宽为 16 位，则它的机器字长为 16 位，也称其为 16 位计算机。通常，机器字长越长，计算机的运算能力越强，其运算精度也越高。

（2）速度。CPU 的速度是指单位时间内能够执行指令的条数。速度的计算单位不一，若以单字长定点指令的平均执行时间计算，用每秒百万条指令（Million Instructions Per Second，MIPS）作为单位；若以单字长浮点指令的平均执行时间计算，则用每秒百万条浮点运算指令（Million Floating-point Operations Per Second，MFLOPS）作为单位。现在，采用计算机中各种指令的平均执行时间和相应的指令运行权重的加权平均法求出等效速度作为计算机运算速度。

（3）主频。主频又称为主时钟频率，是指 CPU 在单位时间内产生的时钟脉冲数，以 MHz/s（兆赫兹每秒）为单位。由于计算机中的一切操作都是在时钟控制下完成的，因此，对于机器结构相同或相近的计算机，CPU 的时钟频率越高，运算速度也就越快。

2.2.2　Intel 8086/8088 微处理器功能结构

Intel 8088 是一个准 16 位微处理器，内部数据总线是 16 位的，而外部数据总线是 8 位

的，也就是说，在一个总线周期内，与外部只能传递一个字节。Intel 8086 微处理器是由美国 Intel 公司 1987 年推出的一款高性能的 16 位微处理器，其内部总线和外部数据总线均为 16 位，即在一个总线周期内能传送一个字。另外，8086 的指令队列为 6 个字节，而 8088 的指令队列为 4 个字节。但是 Intel 8086 和 Intel 8088 的指令系统、指令编码格式和寻址方式是完全相同的，体系结构也是相似的，所以它们在软件设计方面是完全兼容的。

从功能上，8086 CPU 可以分为并行工作的两大部分：执行部件（Execution Unit，EU）和总线接口部件（Bus Interface Unit，BIU）。在执行指令的过程中，两个部件形成了两级流水线：执行部件执行指令的同时，总线接口部件完成从存储器中预取后继指令的工作，使指令的读取与执行可以部分重叠，从而提高了总线的利用率。Intel 8086/8088 CPU 的功能结构如图 2.2 所示。

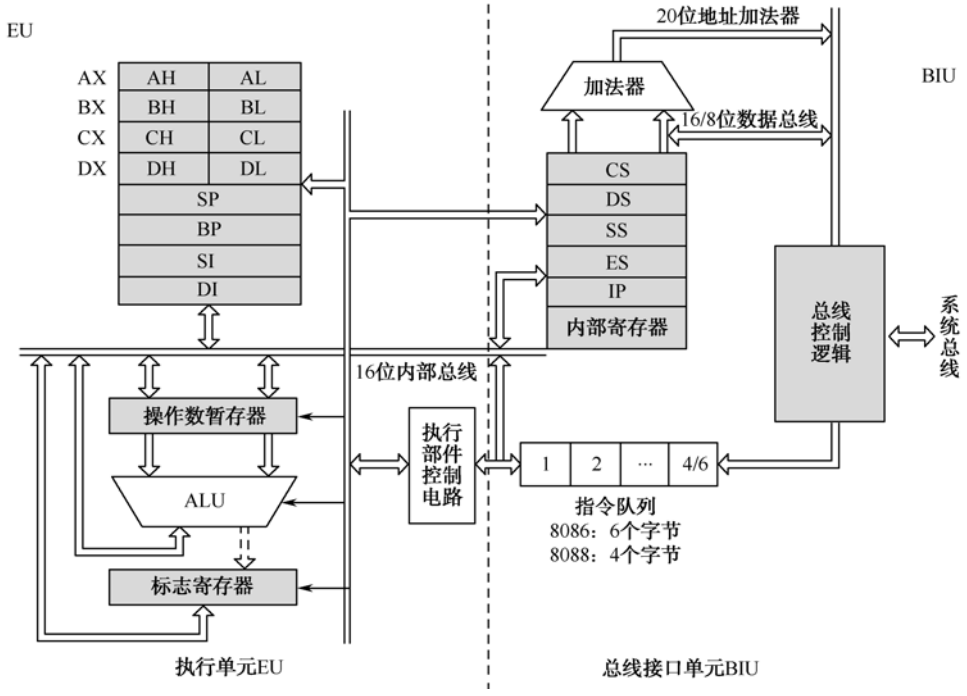


图 2.2　Intel 8086/8088 CPU 功能结构

1. 执行部件（EU）

EU 负责全部指令的分析执行，并管理通用寄存器和标志寄存器。

（1）EU 的重要组成器件

执行部件是由算术逻辑运算单元、通用寄存器、标志寄存器、运算寄存器和执行部件控制电路等组成的。

① 算术逻辑运算单元（ALU）。ALU 是一个 16 位的运算器，可以用于 8 位、16 位二进制数的算术和逻辑运算，也可以按照指令的寻址方式计算访问存储器所需要的 16 位偏移量。

② 通用寄存器组。通用寄存器组包含 4 个数据寄存器（AX、BX、CX、DX）、2 个指针寄存器（BP、SP）和 2 个变址寄存器（SI、DI）。其中数据寄存器既可以作为 16 位的寄存器使用，也可以作为 8 位寄存器使用。

③ 标志寄存器。标志寄存器是一个 16 位的寄存器，每一位记录了 CPU 的运算状态

和某些控制标志。

④ 运算寄存器。运算寄存器的主要作用是协助 ALU 完成运算，暂存运算结果。

⑤ 执行部件控制电路。执行部件控制电路负责从 BIU 指令缓冲队列中取指令，并对指令进行译码，根据指令的要求向 EU 内部各器件发出控制命令。

（2）EU 的工作过程

EU 的工作过程如下：

① 从 BIU 指令队列寄存器中读取指令；

② EU 控制电路对指令进行分析译码，并指出本次操作的性质及操作对象；

③ 计算出指令操作数的 16 位有效地址送 BIU，由 BIU 加法器形成 20 位的物理地址；

④ 将操作数经系统总线和内部数据总线送至 ALU 进行运算；

⑤ 将运算结果经内部数据总线传送并保存在通用寄存器中或通过 BIU 送入相应的存储单元；

⑥ 将反映本次操作的状态标志存放在标志寄存器中。

2. 总线接口部件（BIU）

BIU 的主要功能是负责 CPU、存储器及输入/输出设备的信息传输。BIU 根据段寄存器（CS）和指令指针（IP）形成的 20 位物理地址从存储器中取出指令，并暂存在指令队列中，等待 EU 取走并执行。当 EU 发出访问存储器或输入/输出设备端口的命令时，BIU 根据 EU 提供的数据和地址，进入系统总线周期，存取数据。

（1）BIU 单元的组成

BIU 单元由 4 个 16 位的段寄存器（CS、DS、ES、SS）、一个指令指针寄存器（IP）、一个指令缓冲队列、一个形成 20 位物理地址的加法器，以及总线控制逻辑电路等组成。

① 段寄存器和加法器。Intel 8086 CPU 所有的段寄存器都是 16 位的，它们的任务是负责存放指令和数据段的地址部分。其中，DS 存放数据段的段地址，ES 存放附加数据段的段地址，SS 存放堆栈段的段地址，CS 存放代码段的段地址。在 BIU 中，由段寄存器和加法器形成存储单元的物理地址。Intel 8086 CPU 的物理地址有 20 位，地址加法器将 16 位的段地址内容左移 4 位，并与 16 位偏移地址相加，从而形成 20 位的物理地址。

② 16 位指令指针寄存器 IP。指令指针寄存器是一个 16 位的指针寄存器，它用来存放下一条要执行的指令在代码段中的偏移地址。在取指令的过程中，代码段寄存器和指令指针寄存器一起决定指令在存储器中的物理地址，并且在每次取出一条指令后 IP 自动变成下一条指令的偏移地址。

③ 指令队列缓冲寄存器。Intel 8086 指令队列缓冲寄存器大小是 6 个字节，指令队列缓冲寄存器按照先进先出的方式进行工作，并按顺序取到 EU 中执行。

④ 总线控制逻辑电路。总线控制逻辑电路的主要功能是将 Intel 8086 的内部总线和外部总线相连，是 8086 CPU 与存储器或 I/O 端口进行数据交换的必经之路。总线控制逻辑电路包括 16 条数据总线、20 条地址总线和若干条控制总线。CPU 通过总线控制电路与外部取得联系，从而构成各种规模的 8086 计算机系统。

（2）BIU 的工作过程

① 当指令队列寄存器有两个以上指令字节空闲时，BIU 自动地按 CS 值和 IP 值由地址加法器成的 20 位物理地址到存储器中取指令，并把指令存放在指令队列中。

② 当执行转移指令时，BIU 首先清空指令缓冲队列寄存器，然后从转移后的当前地址取出指令送 EU 执行，最后从主存（主存储器）中取出后继指令送指令队列排队。

③ BIU 根据 EU 的请求，形成访问存储器或输入设备的地址，从而得到所需要的操作数。

④ 取来的操作数经总线控制逻辑传送到内部的 ALU 数据总线，然后再由 EU 完成相应的操作。

⑤ 若需存放操作结果，则由 EU 向 BIU 提出请求，并由 BIU 形成目标地址，将操作结果写入存储器或输出设备中。

从以上对 EU 和 BIU 的工作过程分析可知，EU 和 BIU 是两个既分工又合作的独立单元。BIU 首先通过内部通信寄存器检测是否有 EU 的请求，然后实现 CPU 与存储器或 I/O 设备之间的数据存取操作。在无请求的情况下，BIU 自动地从存储器中读取两个字节的指令，并存放于指令队列中供 EU 使用。在 BIU 取指令的同时，EU 也可以执行指令，这样就形成了 BIU 和 EU 并行工作的状态，从而减少了 CPU 为取指令而等待的时间，提高了 CPU 的利用率，加快了系统运行的速度，因此 Intel 8086/8088 CPU 的指令流水线结构设计的优越性就得到了充分的体现。

2.3 Intel 8086/8088 CPU 寄存器组

寄存器是 CPU 内部存放数据的地方，Intel 8086/8088 CPU 共有 14 个 16 位寄存器。根据其功能，可以分为通用寄存器、段寄存器和控制寄存器。其中，通用寄存器有 8 个，段寄存器有 4 个，控制寄存器有 2 个。Intel 8086 CPU 寄存器都可以作为 16 位的寄存器使用，部分寄存器也可以作为两个 8 位寄存器使用。Intel 8086/8088 CPU 寄存器如图 2.3 所示，由于它们处于 CPU 内部，所以存取速度比存储器要快得多，可以用这些寄存器存放计算过程中所需要的各种信息或计算后得到的结果。

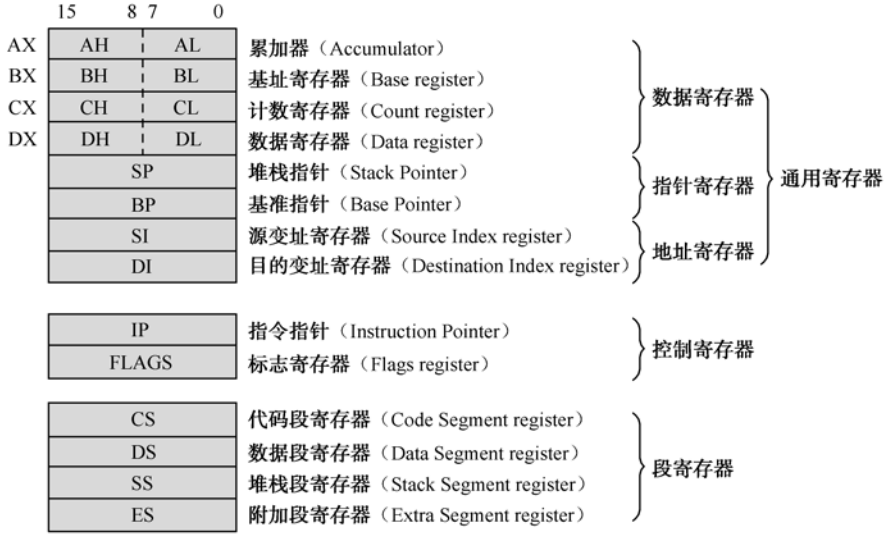


图 2.3 Intel 8086/8088 CPU 寄存器

2.3.1　通用寄存器

在计算机中，通用寄存器通常用来存放一般性的数据，Intel 8086/8088 通用寄存器共有 8 个，分别为 AX、BX、CX、DX、SP、BP、SI 和 DI。可以将它们划分为 3 种，AX、BX、CX 和 DX 为数据寄存器，SP 和 BP 为指针寄存器，SI 和 DI 为变址寄存器。

1. 数据寄存器

数据寄存器有 4 个，分别是 AX、BX、CX 和 DX，主要用来存放算术运算或逻辑运算的操作数和中间结果。这 4 个寄存器的每一个既可以作为一个 16 位寄存器使用，又可以分成两个独立的 8 位寄存器使用。AX 寄存器可以分为 AH 和 AL，BX 寄存器可以分为 BH 和 BL，CX 寄存器可以分为 CH 和 CL，DX 寄存器可以分为 DH 和 DL。

以 AX 寄存器为例，AX 寄存器既可以作为一个 16 位的寄存器使用，又可以作为两个独立的 8 位寄存器使用，其中 AH 是 AX 的高 8 位，AL 是 AX 的低 8 位，如图 2.4 所示。数据寄存器的双重性使得 Intel 8086/8088 非常容易处理字数据和字节数据。

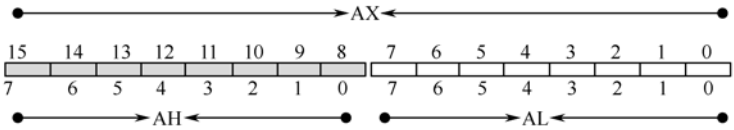


图 2.4　AX 寄存器可以作为两个 8 位寄存器使用

数据寄存器除了暂存数据以外，在一些指令中，它们还具有规定的用途。

（1）AX 又被称为累加器，在乘法和除法指令运算中，用来存放被乘数和被除数。在 I/O 指令中用来存放 CPU 和 I/O 端口之间传递的信息。

（2）BX 又被称为基址寄存器，在访问主存（主存储器）时可以用来存放存储单元的偏移量。

（3）CX 被称为计数寄存器，在循环指令中存放循序的次数，在移位指令中用来存放移动的次数，在具有重复前缀的串操作指令中，用来存放重复串操作的次数。

（4）DX 寄存器在 I/O 指令中，当端口号大于 255 时，DX 用来存放端口号。在 DOS 功能调用中，DX 有时用来存放操作数的值或操作数在内存中的偏移量。

2. 指针寄存器

指针寄存器包括堆栈指针寄存器（SP）和基址指针寄存器（BP），它们都是 16 位的地址指针寄存器。这两个指针寄存器除了可以和数据寄存器一样参加算术或逻辑运算，还主要用在堆栈操作中存放偏移量。

SP 用于存放当前堆栈的栈顶单元的偏移量，即栈顶的相对位置，它始终指向栈顶。当进行堆栈操作（入栈或出栈）时，隐含使用的就是堆栈指针 SP。SP 寄存器和 SS 堆栈段寄存器一起定位堆栈中栈顶数据的物理地址。

BP 与 SP 完全不同，它用于存放当前堆栈内某个单元的偏移量，可以用它对堆栈内任意位置的数据进行操作，它不一定要指向栈顶。BP 寄存器也可与 SS 寄存器联合使用以确定堆栈段中的存储单元地址。

需要特别指出的是，堆栈是主存（主存储器）中一个特殊的区域，它采用先进后出或后进先出的操作方式，而不是随机存取操作方式。堆栈通常由处理器自动维持，由堆栈段寄存器 SS 和指针寄存器 SP 共同指示。所以，SP 不应该再用于其他目的，实际上可归类为专用寄存器。

3. 变址寄存器

变址寄存器包括源变址寄存器（SI）和目的变址寄存器（DI），它们都是 16 位的，通常也用做地址指针。这两个变址寄存器除了可以和数据寄存器一样参加算术或逻辑运算，还主要用在数据的串操作指令中，通常，SI 用来存放源数据串的首地址，DI 用来存放目的串的首地址，并且 SI 和 DI 的作用不能互换。

2.3.2 段寄存器

在 8086/8088 系统中，访问存储器的地址由段地址和段内偏移地址或有效地址 EA（简称偏移量）两部分组成。而段寄存器就是用来存放各分段的逻辑段基值的寄存器，并指示当前正在使用的各个逻辑段。段寄存器是由 Intel 8086/8088 CPU 将存储器分成的若干个段，分别存放程序的代码、数据等，如图 2.5 所示。段寄存器包括代码段寄存器（Code Segment，CS）、数据段寄存器（Data Segment，DS）、堆栈段寄存器（Stack Segment，SS）和附加段寄存器（Extra Segment，ES）4 个段寄存器。

（1）代码段寄存器：存放当前正在运行的程序代码所在段的段基值，表示当前使用的指令代码可以从该段寄存器指定的存储器段中取得，相应的偏移量则由 IP 提供。

（2）数据段寄存器：指出当前程序使用的数据所存放段的最低地址，即存放数据段的段基值。

（3）堆栈段寄存器：指出当前堆栈的底部地址，即存放堆栈段的段基值。

（4）附加段寄存器：指出当前程序使用附加数据段的段基址，该段是串操作指令中目的串所在的段。

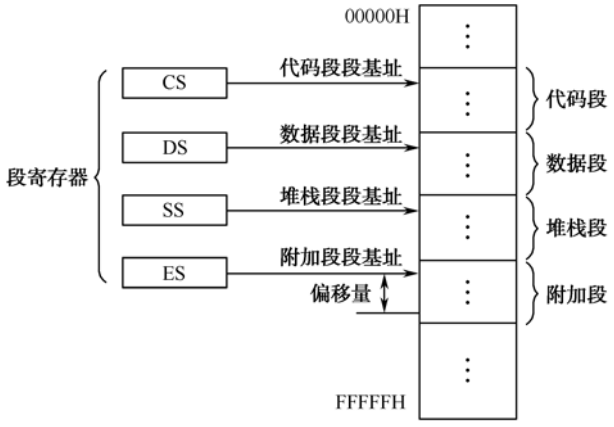


图 2.5　存储器的段结构和段寄存器

一个程序可以被划分成任意个逻辑段，但是在程序运行的任意时刻，最多只能有 4 个当前段。Intel 8086/8088 地址总线为 20 位，最大寻址空间为 1MB，对于 1MB 的内存空间，

最多可以划分为 64K 个物理段，每一个段的最大容量为 64KB。

在汇编语言程序设计中，程序段各段的首地址存放在段寄存器中，其中，CS 用来存放程序指令序列的起始地址，8086/8088 微处理器利用 CS：IP 取得下一条要执行的指令；SS 存放堆栈段的段地址，利用 SS：SP 存取堆栈段中的数据；DS 和 ES 存放当前程序运行所需要数据的段地址。

在汇编语言程序设计中，由于当前段最多只能有 4 个，程序可以通过更换相应的段寄存器的内容在程序运行过程当中调换当前段。例如，某个程序已设置了两个数据段 DATA1 和 DATA2，当数据段寄存器指向 DATA1 的段基址时，DATA1 为当前数据段，如果现在程序要求访问数据段 DATA2 中的某个存储单元，这时就要调换当前段，要用几条指令让 DS 指向 DATA2 的段基址，即将 DS 的内容换成 DATA2 的段基址，使 DATA2 成为当前段。

注意：段基值是一个段起始单元地址的高 16 位，它存放在某一个段寄存器中。段基址是一个段起始单元的 20 位物理地址；偏移量是某个存储单元距离所在段的段基址的字节数。偏移量为 0 时，则这个存储单元就是这个段的起始单元。

2.3.3　控制寄存器

Intel 8086/8088 微处理器有两个控制寄存器：指令指针（Instruction Pointer，IP）和标志寄存器（FLAGS）。

1. 指令指针

指令指针是一个 16 位的寄存器，IP 寄存器存放的是指令在内存空间中的偏移量，它表示的是当前指令的下一条指令的有效地址。IP 与 CS 相结合共同确定下一条指令在存储器上的物理地址。在程序运行期间，CPU 自动修改 IP 的值，使它始终指向正在执行指令的下一条指令代码的起始存储位置，如图 2.6 所示。

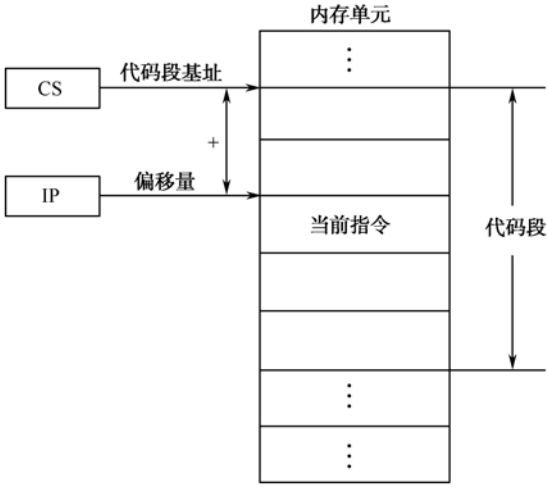


图 2.6　指令指针 IP

指令指针寄存器只能通过指令改变其内容，程序员不能直接存取 IP 的内容。在顺序程序中，注意 CPU 每取一个字节后 IP 内容加 1，但每取一个字后 IP 内容加 2。若要改变

程序的执行顺序，可以通过一些指令间接更改 IP 的值。例如，执行 JMP、JE 等转移指令时，CPU 会将目的地址的偏移量送入 IP。

2. 标志寄存器

Intel 8086/8088 CPU 有一个 16 位的标志寄存器，标志寄存器反映了 CPU 当前的状态和控制。标志寄存器共有 9 个标志位，它们分别是溢出标志位 OF、进位标志位 CF、奇偶标志位 PF、辅助进位标志位 AF、零标志位 ZF、符号标志位 SF、单步标志位 TF、中断允许标志位 IF 和方向标志位 DF。其中，前 6 个标志位是状态标志，它们反映了 CPU 当前的状态，后 3 个标志位是控制标志，控制标志位可用来控制 CPU 执行某些特殊的操作。各个标志位在标志寄存器中的位置如图 2.7 所示。

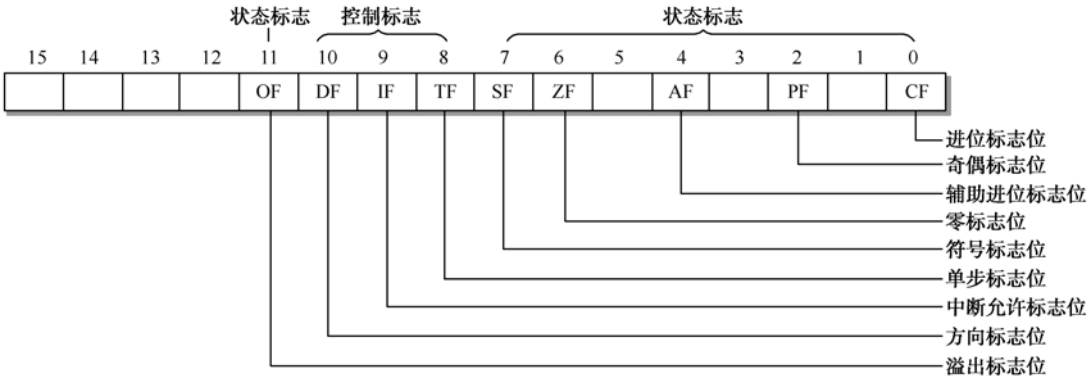


图 2.7　标志寄存器 FLAGS

3. 状态标志

标志寄存器的状态标志位有 6 个，它们分别是 CF、PF、AF、ZF、SF 和 OF。

（1）进位标志位 CF（Carry Flag）

标志寄存器的第 0 位是进位标志位 CF，CF 用来反映运算结果是否产生进位或借位。如果运算结果的最高位（字操作的第 15 位或字节操作的第 7 位）产生一个进位（加法）或借位（减法），则 CF 置 1；否则 CF 清零。另外，CF 在移位/循环移位类指令中用来保存从最高位（左移）或最低位（右移）移出的 0 或 1。

（2）奇偶标志位 PF（Parity Flag）

奇偶标志位 PF 用来反映运算结果低 8 位中“1”的个数的奇偶性，可以为传送数据时可能产生的出错情况提供检验依据。如果运算结果的低 8 位中“1”的个数为偶数，则 PF 置 1；否则 PF 清零。PF 只检查运算结果的低 8 位，与该指令操作数的长度无关。

（3）辅助进位标志位 AF（Auxiliary Flag）

辅助进位标志位 AF 用来反映运算结果低 4 位产生进位或借位的情况。如果运算结果的第 3 位产生了一个进位（加法）或借位（减法），则 AF 置 1；否则 AF 清零。AF 只反映运算结果的低 4 位，与操作数长度无关，主要用于十进制数运算的调整。

（4）零标志位 ZF（Zero Flag）

零标志位 ZF 用来反映运算结果是否为 0。如果运算结果为 0，则 ZF 置 1；否则 ZF 清零。

（5）符号标志位 SF（Sign Flag）

符号标志位 SF 用来反映在将运算结果视为带符号数时，运算结果为正数还是负数。如果运算结果为负，即一个数的最高位（字操作的第 15 位，字节操作的第 7 位）为 1，则 SF 置 1；否则 SF 清零，SF 与运算结果的最高位相一致。

（6）溢出标志位 OF（Overflow Flag）

溢出标志位 OF 用来反映运算结果是否超出了补码的表示范围。8 位带符号数补码的表示范围是−128～+127，16 位带符号数补码的表示范围是−32768～+32767。如果运算结果超过了补码的表示范围，则 OF 置 1；否则 OF 清零。溢出是由两个同符号数相加或异符号数相减产生的，发生溢出时运算结果出错。

4. 控制标志

标志寄存器的控制标志主要负责控制 CPU 完成某些特定的操作，Intel 8086/8088 CPU 标志寄存器的控制标志有 3 个，分别是 TF、IF 和 DF。

（1）单步标志位 TF（Trace Flag）

单步标志位 TF 也称为跟踪位或陷阱标志位，主要用于程序的跟踪调试。当 TF = 1 时，CPU 进入单步方式，每执行一条指令后自动产生一个内部的单步中断，执行相应的中断处理程序，对执行该条指令后的各寄存器情况进行检查；当 TF = 0 时，CPU 正常工作，不进入单步方式，不产生单步中断。TF 常用于程序的调试，如在查错程序 DEBUG 中，可以使用单步命令，在每条指令执行后就停下来进行检查。

（2）中断允许标志位 IF（Interrupt-enable Flag）

中断允许标志位 IF 决定 CPU 是否可以响应外部可屏蔽中断请求。当 IF = 1 时，CPU 为开中断状态，允许 CPU 响应外部可屏蔽中断；当 IF = 0 时，CPU 为关中断状态，不允许 CPU 响应外部可屏蔽中断。编程人员可以通过 STI（IF 置 1）和 CLI（IF 清零）指令来控制 IF 的状态。

注意：IF 只影响外部可屏蔽中断，对非可屏蔽中断和内部中断不起作用。

（3）方向标志位 DF（Direction Flag）

方向标志位 DF 决定串操作指令执行时指针寄存器的调整方向。Intel 8086/8088 CPU 提供了一些串操作指令，串操作有正向处理和反向处理两种方式，当 DF = 0 时，正向处理（从低地址向高地址），串操作指令自动地使变址寄存器（SI 或 DI）递增；当 DF = 1 时，反向处理（从高地址向低地址），串操作指令自动地使变址寄存器（SI 或 DI）递减。编程人员可以通过 STD（DF 置 1）和 CLD（DF 清零）指令来控制 DF 的状态。

【例 2.1】　设（AX）=0010 0011 0100 1101B，（DX）=0101 0010 0000 1001B，试指出两数相加后，6 位标志位的状态。

使用补码公式对两数进行运算，并按定义对结果进行判别。因为计算机中存储的已是补码，所以两数相加过程如下

$$\begin{array}{r} \mathbf{0}\ 010\ 0011\ 0100\ 1101 \\ +\ \mathbf{0}\ 101\ 0010\ 0000\ 1001 \\ \hline \mathbf{0}\ 111\ 0101\ 0101\ 0110 \end{array}$$

根据两数相加结果，可得如下结论：

① 结果非零，所以 ZF=0；

② 低 8 位中共有 4 个 1（偶数个），所以 PF=1；
③ 根据符号位，可知 SF=0；
④ 运算结束后，向更高位无进位，所以 CF=0；
⑤ 运算结果无溢出，所以 OF=0；
⑥ 第 3 位向第 4 位产生进位，所以 AF=1。

2.4 存 储 器

存储器是存放程序和数据的部件，计算机运行所需要的指令和数据都存放在存储器中。存储器可分为主存储器（内存储器）和辅助存储器（外存储器）两种，辅存储器是指磁盘或磁带等设在主机之外的外部设备，主存储器又分为只读存储器和随机存取器。本书所说的存储器一般是指主存储器。

2.4.1 存储器的段结构

Intel 8086/8088 CPU 有 20 条地址线，每条地址线状态为 0 或 1，所以它的寻址空间就是 2^{20} B = 1MB，但是目前的状况是，Intel 8086/8088 CPU 内部的 ALU 只能处理 16 位的地址运算，与地址有关的寄存器也都是 16 位的，16 位最多只能寻址 2^{16}B=64KB。那么，怎样才能让 16 位的寄存器的寻址能力达到 1MB 呢？

为了解决这个问题，Intel 8086/8088 采用了存储器分段的概念，即把 1MB 空间的存储器分成若干个逻辑段，每个逻辑段的大小按实际需要确定，其未用的空间可以定义为其他逻辑段，但最大为 2^{16}B = 64KB。这样 CPU 就可以通过“段地址：段内偏移量”对 1MB 的内存单元进行寻址。

关于存储器段结构的划分需要注意以下几点。

（1）段基址用来说明逻辑段在内存中的存储位置。为了能用 16 位的段寄存器表示段地址，8086/8088 微处理器规定段地址必须能被 16 整除，即段地址是模 16 地址。所以 8086/8088 微处理器最多可以划分 2^{16}=64K 个段。

（2）段内偏移量表示该存储单元的物理地址与该单元所在的逻辑段的物理地址之间的差值。受 16 位寄存器的限制，段内偏移量的最大值为 2^{16}B，即每一个段不超过 64KB。

（3）逻辑段之间的关系多种多样，在物理存储器中可以是首尾相连的，也可以是相互间隔开的，还可以是完全重叠的或部分重叠的。逻辑段之间的关系如图 2.8 所示。

（4）一个存储器虽然可以分为若干个段，但是在任何时刻，一个程序最多只能访问其中的 4 个逻辑段的内容。这 4 个段分别是代码段、数据段、堆栈段和附加段，分别由 4 个段寄存器 CS、DS、SS 和 ES 指向这 4 个逻辑段。如果在程序设计中有多于 4 个的逻辑段，可以通过更换段寄存器的内容，使其指向不同的逻辑段，从而在不同的逻辑段之间完成转换。

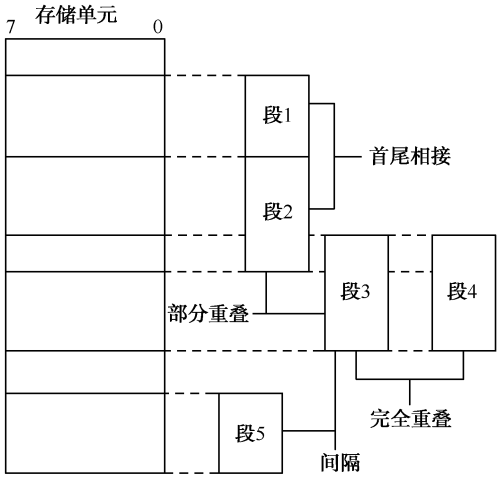


图 2.8　逻辑段之间的关系

2.4.2　物理地址与逻辑地址

Intel 8086/8088 处理器有 20 根地址总线，可以对 1MB 的存储单元进行寻址，其地址范围为 00000H～FFFFFH，这些地址与存储单元一一对应，称为存储单元的物理地址。

Intel 8086/8088 能对 1MB 的内存单元进行寻址，但是 8086/8088 内部寄存器均为 16 位，而且 8086/8088 给出的地址码也只有 16 位。为了解决这个问题，Intel 8086/8088 对存储器采用分段管理，将 1MB 的存储空间分成若干个逻辑段，每个逻辑段的长度不大于 64KB。某个存储单元相对于其所在段起始位置的偏移字节数称为逻辑地址，逻辑地址也称为偏移地址。

Intel 8086 存储器上每个存储单元的地址都有以下两种表示方式。

（1）物理地址，即这个单元在存储器上的地址编号。在 1MB 的存储空间中，每个物理地址都是唯一的，由 20 位的二进制数组成，范围是 00…00（20 位）～11…11（20 位）。CPU 与存储器之间交换信息都是通过物理地址进行的。

（2）逻辑地址，即以“段基值：偏移量”的形式表示存储单元地址。段基值和偏移量都用无符号的 16 位二进制数或 4 位十六进制数表示。段基值是一个段起始单元地址的高 16 位，它存放在某一个段寄存器中。但是 CPU 不能根据逻辑地址访问存储单元，为了访问存储单元，必须首先将逻辑地址转换成物理地址。逻辑地址到物理地址的转换是通过总线接口单元 BIU 来完成的。BIU 首先将存放在某个段寄存器中的逻辑地址的段基值（16 位）左移 4 位，形成 20 位的段基址，再与 16 位的偏移量相加，产生 20 位的物理地址，即：

20 位物理地址=16 位段基值×10H+16 位偏移量

或

20 位物理地址=20 位段基址+16 位偏移量

逻辑地址和物理地址的转换方式可以用图 2.9 表示。

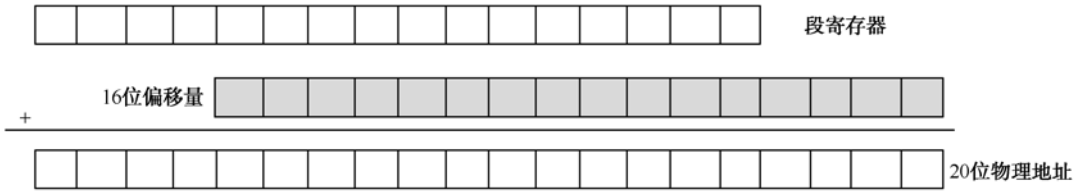


图 2.9　物理地址的形成

【例 2.2】　设某存储单元的逻辑地址为 1234H：5678H，将其转换为物理地址。

因为 16 进制的一位可以表示二进制数的 4 位，所以其转换成物理地址的方法如下：

$$\begin{array}{r} 12340\text{H} \\ +\ \ 5678\text{H} \\ \hline 179\text{B8H} \end{array}$$

【例 2.3】　一个字在存储器中，偏移量为 34H，若 DS=1000H，这个字在存储器中的物理地址是多少？

根据物理地址计算方法，物理地址=16 位段基值×10H+16 位偏移量，所以其物理地址如下：

$$\begin{array}{r} 1000\text{H} \\ \times\ \ \ 10\text{H} \\ \hline 10000\text{H} \\ +\ \ \ 34\text{H} \\ \hline 10034\text{H} \end{array}$$

【例 2.4】　若 CS=2000H，IP 为 123H，程序在执行过程中将取出的下一条指令的物理地址是多少？

根据物理地址计算方法，物理地址=16 位段基值×10H+16 位偏移量，所以其物理地址如下：

$$\begin{array}{r} 2000\text{H} \\ \times\ \ \ 10\text{H} \\ \hline 10000\text{H} \\ +\ \ 123\text{H} \\ \hline 20123\text{H} \end{array}$$

通常情况下，CPU 能根据需要自动选择段寄存器，一般在指令中不指明所需要的段寄存器，取指令和堆栈操作所引用的段寄存器分别规定为 CS 和 SS，是不可变的；串操作中目的操作数的段寄存器规定为 ES，也是不变的。但是在存取一般存储器操作数时，段寄存器不一定是 DS，当偏移量涉及 BP 时，段寄存器也不一定非得是 SS。允许替代也叫段超越，这样段基值除了可以由默认的段寄存器提供以外，也可以由其他段寄存器提供。8086/8088 允许使用段超越前缀，即用段超越前缀直接指定引用的段寄存器，如 ES:[BX]，表示数据放在以 ES 内容为段基值，以 BX 内容为偏移量的存储单元中。如果不指定段寄存器，如[BX]，则表示数据存放在以默认的段寄存器 DS 的内容为段基值，以 BX 内容为偏移量的存储单元中。表 2.2 列出了不同操作类型获得段基值和偏移量来源的使用规定。

表 2.2　段寄存器引用规律

访问存储器涉及的方式	默认的段寄存器	可选用的段寄存器	偏　移　量
取指令	CS	无	IP
堆栈操作	SS	无	SP
串操作的源操作数	DS	CS、ES、SS	SI
串操作的目的操作数	ES	无	DI
以 BP 作基址	SS	CS、DS、ES	有效地址 EA
一般数据访问	DS	CS、ES、SS	有效地址 EA

2.4.3　内存地址空间

计算机的主存储器有两种：只读存储器（Read Only Memory，ROM）和随机存取存储器（Random Access Memory，RAM）。ROM 和 RAM 共同构成了 Intel 8086/8088 的内存地址空间。

RAM 是随机存储器，分别是主随机存储和各种接口卡上的 RAM。

主随机存储器是指装在 PC 机主板上的 RAM 和插在扩展槽上的 RAM，这也是人们常说的内存。主随机存储器主要用于存放供 CPU 使用的绝大部分的程序和数据。

某些接口卡需要对大量的数据进行暂时存储，例如在显卡上的 RAM，一般称为显存。显示卡随时可以将显存中的数据在显示器上输出。

只读存储器 ROM 中装有基本输入/输出系统（Basic Input/Output System，BIOS），BIOS 是由主板和各类接口卡（如显卡，网卡）生产商提供的软件系统。通过 BIOS 可以对硬件进行基本的输入/输出操作。例如，在主板上有 ROM 存储芯片，它存储主板的 BIOS；在显卡和网卡上也有 ROM 芯片，它们分别存储着显卡和网卡的 BIOS。

RAM 和 ROM 存储器，在物理上是相互独立的，但是对 CPU 来说，它们是没有分别的。它们都和 CPU 的总线相连接，并且 CPU 可以通过控制总线对它们进行读/写。也就是说，CPU 把它们加起来看做是有若干存储单元的逻辑存储器，这个逻辑存储器就是人们常说的内存地址空间。计算机内存地址空间如图 2.10 所示。

内存地址空间的大小受 CPU 地址总线宽度的限制，Intel 8086/8088 CPU 的地址总线的宽度为 20 位，所以 8086/8088 PC 地址空间的大小为 1MB。不同的计算机系统内存地址空间的分配情况不一样。图 2.11 展示了 Intel 8086/8088 计算机的地址空间分配情况。

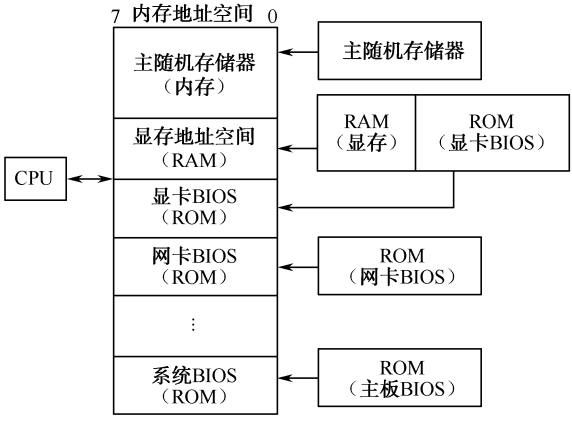


图 2.10　内存地址空间

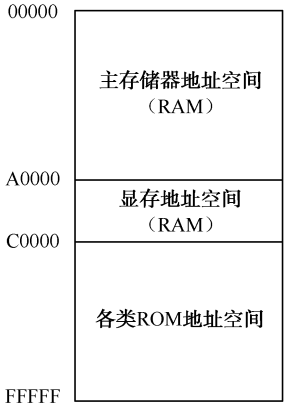


图 2.11　Intel 8086/8088 内存地址空间分配情况

在汇编语言程序设计时，必须知道这个系统的内存地址空间分配情况。因为当读/写某类存储器数据时，必须知道它第一个单元的地址和最后一个存储单元的地址。只有这样才能保证所进行的读写操作在预期的存储器中进行。例如，对于 8086/8088 计算机系统，当向 A0000～BFFFF 写数据时，就是向显存中写入数据。这些数据会被显卡输出到显示器上。而当向 C0000～FFFFF 内存单元中写入数据时，写入数据的操作是无效的。

2.5 堆　　栈

堆栈主要用来暂存数据、子程序调用或中断处理程序的断点地址。在堆栈中，信息的存取严格按照一定的规则进行，在堆栈中对信息的所有操作都在栈顶上进行。堆栈是一种后进先出（Last In First Out，LIFO）的数据结构，堆栈的应用十分广泛，它不仅应用在汇编语言程序设计中，还应用于其他高级语言程序设计和一些现代的工具软件中。

2.5.1 堆栈的概念

堆栈是一个在计算机学科中经常使用的抽象数据类型。从物理上讲，堆栈是一段连续分配的存储空间。在计算机中堆栈的实现形式有两种，一种是按堆栈的工作方式设计的存储器，称为专用堆栈存储器；另一种是由程序设计人员在存储器中划出的一块存储区，称为软件堆栈。现在大多数计算机中通常采用的都是软件堆栈。8086/8088 也采用软件堆栈，如图 2.12 所示。

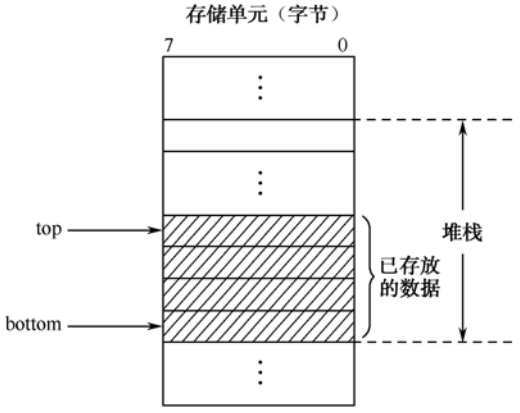


图 2.12　Intel 8086/8088 堆栈结构

堆栈是一种线性表，它的操作有如下几个特性。

（1）堆栈的一端是固定的，另一端是浮动的。固定的一端设置在堆栈的高地址单元，称为栈底（bottom）；浮动的一端可随数据的存取上下浮动，被称为栈顶（top）。

（2）堆栈有两种操作：压栈和出栈，当堆栈中没有数据（即堆栈空）时，栈顶和栈底是重合的。通常把数据存入堆栈的过程称为压栈，把数据从堆栈取出的过程称为出栈。在压栈时，栈顶向低地址单元方向移动。出栈时栈顶向高地址方向移动，栈顶地址随着存入

和取出数据而不断变化。

（3）随着数据的入栈和出栈，数据在堆栈中的位置不断变化。为了指示堆栈中存放数据的位置，Intel 8086/8088 CPU 设置了一个指针寄存器 SP（Stack Point），SP 总是指向堆栈的顶部。

（4）堆栈中对数据的所有操作都是按字操作进行的。

2.5.2　堆栈的组织

堆栈是一块保存数据的连续内存。堆栈既可以向下增长（向内存低地址），也可以向上增长（向内存高地址），增长方向依赖于具体的实现。在 Intel 8086/8088 CPU 中，堆栈是向下增长的，很多计算机都采用这种实现方式，包括 Intel、Motorola、SPARC 和 MIPS 处理器。堆栈指针 SP 也是依赖于具体实现的，它可以指向堆栈的最后地址，也可以指向堆栈之后的下一个空闲可用地址。在 Intel 8086/8088 CPU 中，通过段寄存器 SS 和堆栈指针 SP 的设置，SS 在存储器中指定一个堆栈段，SP 指向堆栈的最后地址，如图 2.13 所示。

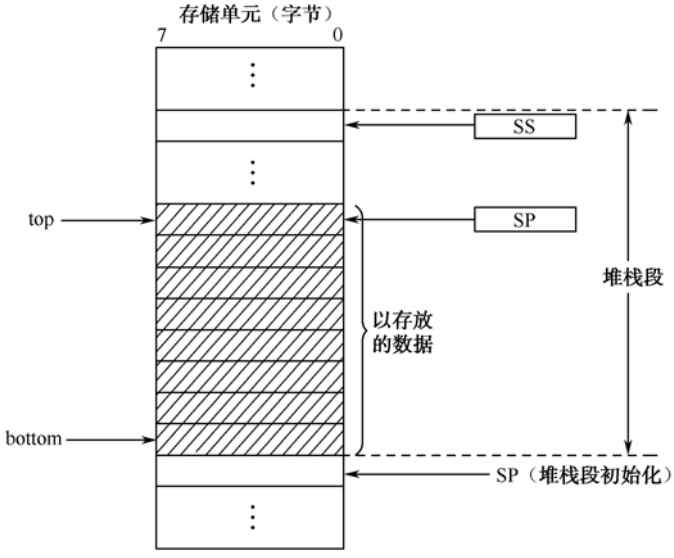
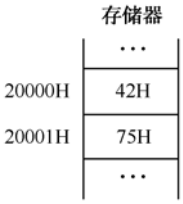


图 2.13　Intel 8086/8088 堆栈组织

堆栈段中所包含的存储单元字节数通常称为堆栈深度或堆栈长度。栈底是堆栈的最大字单元地址，而栈顶则由堆栈指针 SP 指向。SP 中存放着段基址与栈项之间的距离（字节数）。当 SP 初始化时，它指向栈底加 2 字节单元，它的值就是这个堆栈的长度。由于 SP 是一个 16 位的寄存器，因此堆栈长度最大为 2^{16}B = 64KB = 65536B，最多可存放 32K 个字数据。由于 8086/8088 系列微型计算机中一个堆栈的长度是有限的，在编程时可以在程序中设置多个堆栈段，通过重新设置堆栈寄存器 SS 的方法来实现扩大或更换堆栈区的目的。在程序中每次更换堆栈寄存器 SS 后，必须同时对 SP 进行修改，赋予 SP 一个新值。

8086/8088 系列微型计算机的堆栈是按字组织的，即每次在堆栈中存取数据均是两个字节，字数据 7542H 在堆栈中存放的格式如下。



一般存储单元从理论上可以用 SP 加偏移量来引用，而当有字入栈或出栈后，这些偏移量就变了，因此编程时可以使用另一个与堆栈有关的寄存器 BP，BP 不会受到入栈和出栈操作的影响。

2.5.3 堆栈的操作

在汇编语言程序设计中，堆栈的使用非常频繁。在程序设计中堆栈的基本操作有三种：堆栈初始化、入栈和出栈。

1. 初始化堆栈

若进行堆栈操作，必须首先由程序设计人员通过程序在存储器中设置一段区域作为堆栈段。堆栈段中包含的存储单元的个数（字节）是堆栈的深度（堆栈长度）。利用 SP 作为堆栈的栈顶指示器，通过程序指令使它指向栈底加 2 的位置，这种操作称为初始化堆栈。

堆栈初始化主要是对 SS 和 SP 进行赋值。通常在源程序中定义一个段作为堆栈段的语句如下：

```
        ⋮
STACKl  SEGMENT  PARA  STACK
DW      100      DUP (0)
STACKl  ENDS
        ⋮
```

其中，被定义的堆栈段的名字为 STACK1，第 1 行最后的“STACK”说明本段是作为堆栈段使用的，第 2 行的 100 是被定义的堆栈段的长度。当程序经过汇编、连接、装入内存以后，系统会自动分配一个存储区域作为堆栈段，并把这个段的段基址高 16 位送入 SS，同时把堆栈长度 100（64H）送入 SP，作为 SP 的初始值。这样一个堆栈段就被定义并初始化了，程序就可以使用它了。

2. 入栈

将数据压入堆栈的操作称为入栈。入栈操作既可以用指令 PUSH 将通用寄存器、段寄存器或字存储单元的内容压入堆栈顶部，也可以用 PUSHF 指令将标志寄存器的内容压入堆栈，还可以由机器自动实现。

入栈操作是在堆栈顶部进行的，执行过程如下。

（1）修改栈顶，将 SP 减 2。

（2）存入数据，将寄存器或字存储单元的内容送入 SP 所指向的字单元。

图 2.14 表示了将 AX 和 BX 寄存器内容入栈的过程。

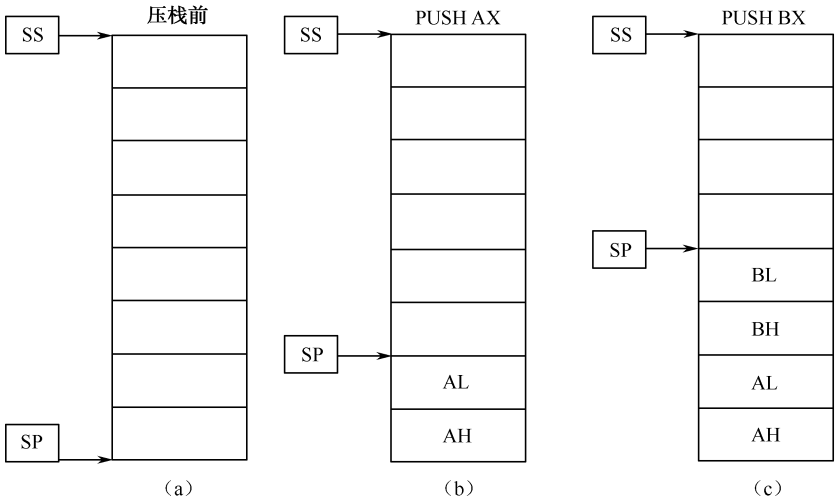


图 2.14　入栈操作

3. 出栈

将数据弹出堆栈的操作称为出栈。出栈操作既可以用指令 POP 将堆栈顶部的一个字数据送入通用寄存器、段寄存器或字存储单元，也可以用 POPF 指令将栈顶的一个字数据送入标志寄存器，还可以由机器自动实现。

堆栈的规则是“后进先出”，所以出栈操作也是在堆栈顶部进行的，其操作过程如下。

（1）取出数据，将 SP 所指向的字单元内容送入寄存器或字存储单元。

（2）修改栈顶位置，将 SP 加 2。

图 2.15 表示了将 BX 和 AX 寄存器出栈的过程。

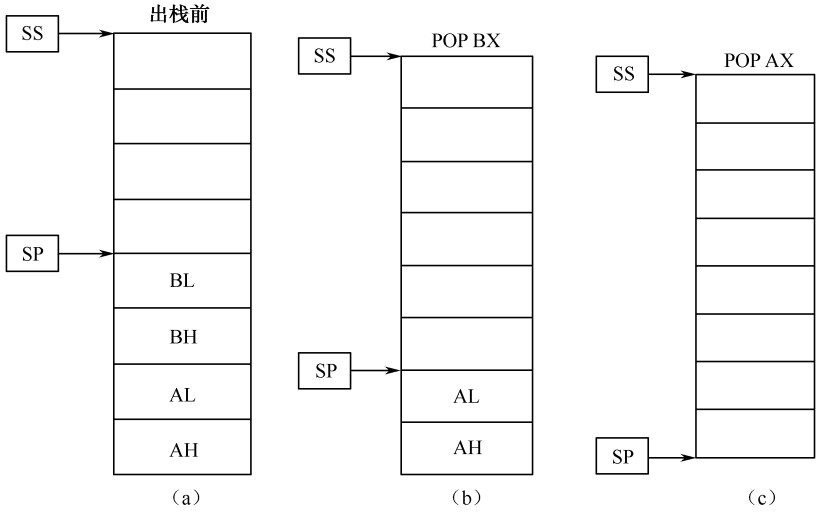


图 2.15　出栈操作

2.6 输入/输出设备

输入/输出（Input/Output）设备，简称 I/O 设备，也称为外部设备，是提供计算机系统与外部世界交换信息的设备总称，是计算机与用户或其他设备交互的桥梁。计算机用户通过外部设备与计算机进行信息交互。用户使用输入设备把程序和数据输入计算机，程序运行的结果又通过输出设备输出给用户。显而易见，输入/输出设备是计算机的重要组成部分之一。直接对输入/输出设备进行控制是汇编语言程序设计最复杂的部分之一。

习题

1．简述计算机系统的组成及各部分的作用。

2．简述 8086/8088 CPU 的组成部分及它们的功能。

3．8086/8088 CPU 所有的寄存器分为几组？每组都有哪些寄存器？每个寄存器有哪些特殊的用途？

4．已知（BH）= 23H，（BL）= 45H，能不能说（BX）= 2345H？SI 寄存器是否可以分为 SH 和 SL？

5．8086/8088 CPU 中标志寄存器各个标志位有什么含义？状态标志和控制标志有什么区别？画出标志寄存器。

6．8086/8088 有哪四个逻辑段？各个逻辑段分别有什么用途？

7．简述 8086/8088 CPU 中所有的段寄存器的作用及汇编语言程序设计中的段结构。

8．什么是逻辑地址？什么是物理地址？如何由逻辑地址转换成物理地址？

9．物理地址为 01000H 和 01004H 处分别放 1ABEH 和 23FCH，用图画出它们的放置情况。

10．求 62A0H+1234H 的和，并根据结果设置各个标志位。

11．在 8086/8088 系统中，某存储单元的逻辑地址如下，试计算出它们的物理地址。

（1）2345H:0112H

（2）2C00H:0l01H

（3）2350H:1234H

（4）300DH:0131H

12. 数据段中某个存储单元在（DS）= 0C07H 时，物理地址是 0C270H，当（DS）= 0C00H 时，物理地址为多少？

13．已知一条指令执行前，（CS）= 0C08H，（IP）= 200H，该指令的第一个字节的物理地址是多少？

14．设堆栈寄存器（SS）= 3E4BH，程序中设置堆栈长度为 300 字节，试计算出堆栈底字单元物理地址、SP 初始值（即堆栈中没有数据时）和 SP 初始值指向的物理地址。

15．设（AX）= 1234H，（BX）= 5678H，（CX）= 100H，（SP）= 1000H，试分析以下程序片段执行后 AX、BX、CX、SP 中的内容分别为多少，并画出堆栈变化的示意图。

```
PUSH    AX
PUSH    BX
PUSH    CX
POP     AX
POP     CX
POP     BX
```

第 3 章

汇编语言程序及其操作过程

机器语言是机器指令的集合，而汇编语言却是机器语言的符号化的表示。汇编语言程序设计就是用汇编语言设计程序，完成指定应用功能。汇编语言是一门实践性极强的课程，必须时刻注意理论结合实践。本章将借助一些工具软件帮助读者掌握一些学习汇编语言必须掌握的概念，如机器语言程序、汇编语言程序，以及汇编语言程序的编译、连接和运行等知识，同时简要介绍这些工具软件的基本用法。

3.1 机器语言程序

机器指令是 CPU 能直接识别并执行的指令，它的表现形式是二进制编码。机器指令通常由操作码和操作数两部分组成，操作码指出该指令所要完成的操作，即指令的功能，操作数指出参与运算的对象，以及运算结果所存放的位置等。

由于机器指令与 CPU 紧密相关，所以，不同种类的 CPU 所对应的机器指令也就不同，而且它们的指令系统往往相差很大。但对同一系列的 CPU 来说，为了满足各型号之间具有良好的兼容性，新一代 CPU 的指令系统必须包括先前同系列 CPU 的指令系统。只有这样，先前开发出来的各类程序在新一代 CPU 上才能正常运行。

机器语言由机器指令组成，CPU 能直接执行用机器语言描述的程序。

3.1.1 机器语言程序

机器语言程序就是由机器语言组成的序列。下面举一个机器语言程序例子。

【例 3.1】 在显示器上输出字符串“Hello,World”。

机器语言程序十六进制代码如下：

```
0B4A:0200       48                  ; H
0B4A:0201       65                  ; e
0B4A:0202       6C                  ; l
0B4A:0203       6C                  ; l
0B4A:0204       6F                  ; o
0B4A:0205       2C                  ; ,
0B4A:0206       57                  ; W
```

```
0B4A:0207      6F              ; o
0B4A:0208      72              ; r
0B4A:0209      6C              ; l
0B4A:020a      64              ; d
0B4A:020b      24              ; $
0B4A:0100      B409            ; MOV AH,09
0B4A:0102      BA0002          ; MOV DX,0200
0B4A:0105      CD21            ; INT 21
0B4A:0107      CD20            ; INT 20
```

程序分析如下。

代码中“;”号后面是注释。其中加黑的部分是机器语言程序，从内存 0B4A:0100 开始存放，后面注释部分是该机器语言的汇编语言；上面没有加黑的部分是字符串“Hello,World”的 ASCII 码，是机器语言程序运行时需要显示的数据，从内存 0B4A:0200 开始存放。

如果需要在显示器上输出字符串“Good Bye!”，仅需要改变显示数据即可。

从上面例子可以看出，机器语言程序为二进制代码，所以编写困难、难懂，并且不易排错。因此，除了在计算机发展的早期，程序员不得不用机器语言开发程序，以及现在的一些特殊情况下采用机器指令外，目前几乎没有人采用机器语言编写程序了。

3.1.2　机器语言程序的输入和运行

Debug 是一款随手可得，直观易用的调试工具，能够实现机器语言程序的输入、调试和运行。

【例 3.2】　采用 Debug 调试工具，使用机器语言程序在显示器上输出字符串“Hello,World”。

步骤如下。

1. 运行 Debug 调试工具

在 Windows 下，选择【开始】→【运行】命令，弹出“运行”对话框，如图 3.1 所示，在“运行”对话框中输入“debug”，单击“确定”按钮进入 DOS 环境下的 Debug 窗口，如图 3.2 所示。



图 3.1　运行对话框



图 3.2　Debug 窗口

2. 使用 E 命令向内存中写入机器码

在 Debug 窗口中输入“E200”后单击回车键，如图 3.3 所示，再分别输入“Hello ,World”

的 ASCII 码“48”、“65”、“6C”、“6C”、“6F”、“2C”、“57”、“6F”、“72”、“6C”、“64”和“24”，如图 3.4 所示。其中，每输入一个 ASCII 码后，单击空格，输入下一个 ASCII 码，输入结束后按回车键。

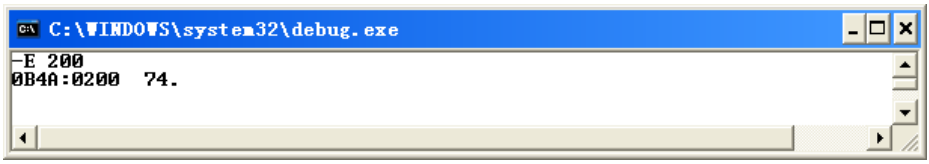


图 3.3　输入 E 命令

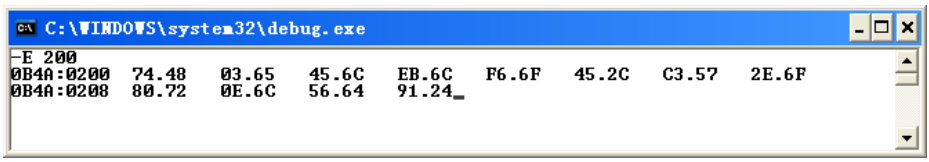


图 3.4　在内存中输入机器码

3. 使用 E 命令向内存中写入机器语言程序

在 Debug 窗口中输入“E 100”后回车，再输入显示“Hello,World”的机器语言程序，分别输入“B4”、“09”、“BA”、“00”、“02”、“CD”、“21”、“CD”和“20”，如图 3.5 所示，输入方式和步骤 2 相同。

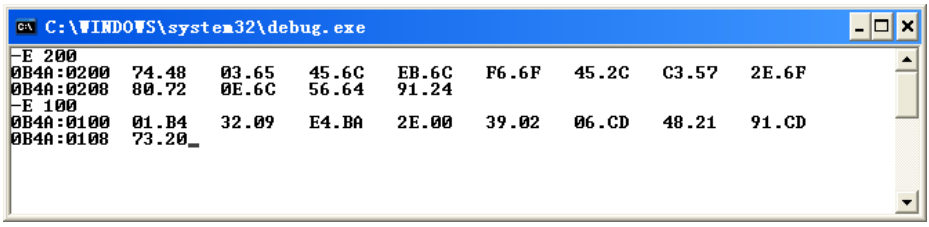


图 3.5　在内存中输入机器语言程序

4. 使用 G 命令运行程序

在 Debug 窗口中输入“G”命令后按回车键，运行的结果如图 3.6 所示，在 Debug 窗口中显示出了“Hello,World”。

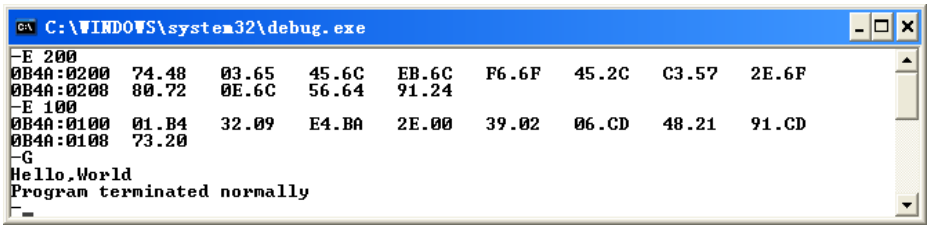


图 3.6　使用 G 命令运行程序

3.1.3　Debug 调试工具

在上一节使用了 Debug 调试工具输入机器语言程序，运行此程序显示“Hello,World”

字符串。Debug 是 DOS、Windows 都提供的实模式（8086 方式）程序的调试工具。使用它可以查看 CPU 各种寄存器中的内容、内存的情况和在机器码级跟踪程序的运行。

Debug 的命令比较多，共有 20 多个，常用到的 Debug 命令如下。

- R 命令：查看、修改 CPU 寄存器的内容。
- D 命令：查看内存中的内容。
- E 命令：修改内存中的内容。
- U 命令：将内存中的机器指令翻译成汇编指令。
- T 命令：跟踪与显示。
- A 命令：以汇编指令的格式在内存中写入机器指令。

下面对各个 Debug 命令进行详细介绍。

1. 使用 D 命令查看内存中的内容

D 是 Display 的头一个字母，D 命令用于显示内存中内容。有以下三种格式。

（1）使用 D 命令格式一方式查询内存的内容

格式：

D

进入 Debug 后，在命令提示符“-”后，输入 D 命令，按回车键。将在屏幕上显示当前段，位移为 100H 起始的 128 个字节内容，如图 3.7 所示，其中每行显示 16 个字节的内容。屏幕左侧为起始地址“段地址：位移”，中间以十六进制形式显示 16 个字节的内容，前 8 个字节和后 8 个字节用“-”分割以便于阅读。右侧以 ASCII 码字符形式显示这 16 个单元的内容，如果该单元内容为不可显示字符，则以小数点的形式显示。

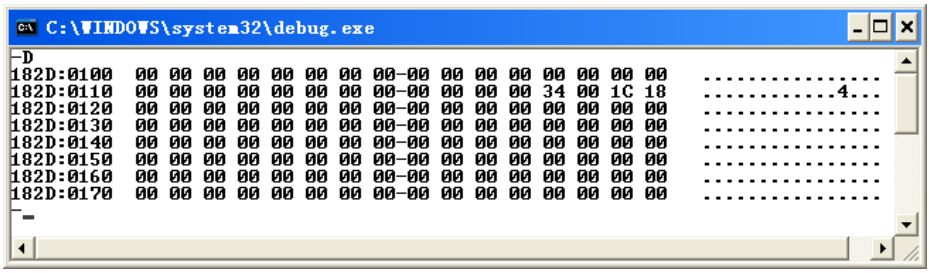


图 3.7　使用 D 命令格式一方式查询内存的内容

注意：

Debug 的命令不区分大小写，输入“D”和“d”效果是相同的。

使用 D 命令格式一方式查询内存的内容，会显示 100H 偏移地址处开始的内存内容，基地址取自 CS 寄存器，之后将会接着上一次显示的内容继续显示，每次都是 128 个字节。

（2）使用 D 命令格式二方式查询内存的内容

格式：

D　[地址]

格式一方式的 D 命令仅能从当前代码段位移为 100H 处开始显示，或者是继前一次后显示紧跟着的 128 个单元内容。当需要更快捷地直接显示内存中未指定地址处开始的内容时，可以使用 D 命令第二种格式查询。

进入 Debug 后，在命令提示符“-”后输入“D 0101:0001”，按回车键，查询结果如

图 3.8 所示。右边是每个内存单元中的数据对应的可显示的 ASCII 码字符。例如，内存单元 0101:004D、0101:005D、0101:006D 和 0101:006E 中存放的数据是 68H、4FH、38H 和 4AH，它对应的 ASCII 字符分别是“h”、“O”、“8”和“J”；内存单元 0101:0010 中的数据是 1DH，它没有对应可显示的 ASCII 字符，Debug 就用“.”代替。

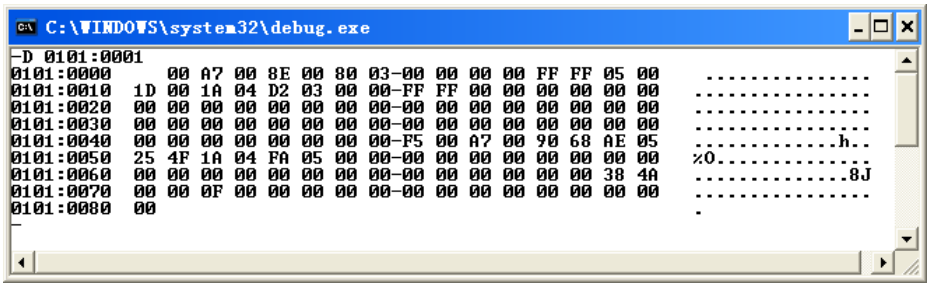


图 3.8　使用 D 命令格式二方式查询内存的内容

如果要指定显示内存单元的个数，直接在命令地址后输入一个空格，再输入要显示的内存单元个数，如在命令提示符“-”后输入“D 0101:0001 4”，按回车键，结果如图 3.9 所示。

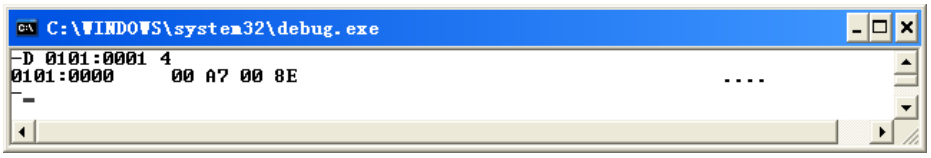


图 3.9　使用 D 命令格式二方式查询指定内存和个数的内容

注意：运行程序后用户看到的内存中的内容，在不同的计算机中是不一样的，也可能每次用 Debug 后看到的内容都不相同，因为用户用 Debug 看到的都是原来就在内存中的内容，这些内容受随时都有可能变化的系统环境的影响。当然，用户也可以改变内存和寄存器中的内容。

（3）使用 D 命令格式三方式查询内存的内容

格式：

D　[地址范围]

使用 D 命令格式三方式可以很方便地观察限定在一定范围的内存内容。

进入 Debug，在命令提示符“-”后输入“D 101 104”，按回车键，查询结果如图 3.10 所示，表示显示当前段，位移从 101H 开始到 104H 结束的内存内容。

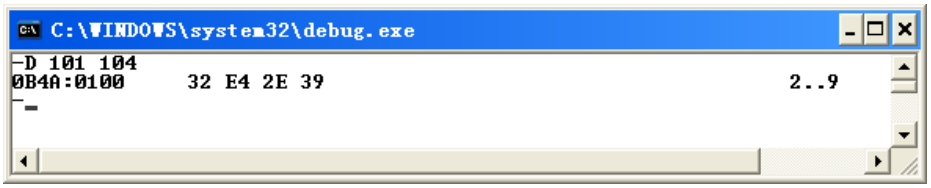


图 3.10　使用 D 命令格式三方式查询内存的内容

2. 使用 E 命令修改内存中的内容

E 是 Edit 的头一个字母，E 命令用于编辑修改内存中内容。

格式：

E　[地址]

可以使用 E 命令向内存中写入数值。例如，用 E 命令在内存 1000:0～1000:9 中分别写入数值 0、1、2、3、4、5、6、7、8、9，方法是：在命令提示符“-”后输入“E 起始地址数据数据……”的格式，输入完后回车。使用 D 命令查询，结果如图 3.11 所示。

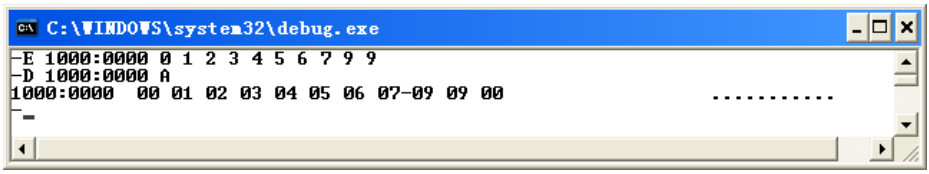


图 3.11　使用 E 命令向内存中写入数值

也可以使用 E 命令向内存中写入字符。例如，用 E 命令在内存 1000:0～1000:A 中分别写入字符 H、e、l、l、o、,、W、o、r、l、d，方法是：在命令提示符“-”后输入“E 起始地址 ‘字符’‘字符’……”的格式，输入完后回车。使用 D 命令查询，结果如图 3.12 所示。从图中可以看出，Debug 对 E 命令的执行结果是，向内存单元中分别写入各个字符的 ASCII 码值。

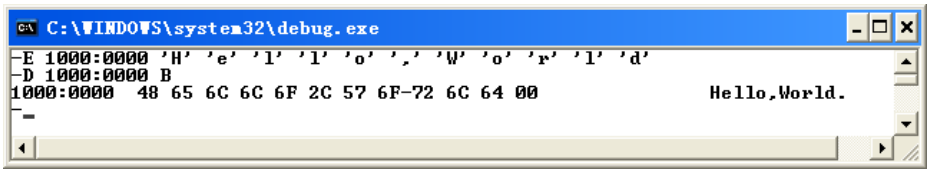


图 3.12　使用 E 命令向内存中写入字符

当然，还可以使用 E 命令向内存中写入字符串，例如，用 E 命令从内存 1000:0 处开始写入字符串“Hello,World”，方法是：在命令提示符“-”后输入“E 起始地址‘字符串’”的格式，输入完后回车，使用 D 命令查询，结果如图 3.13 所示。

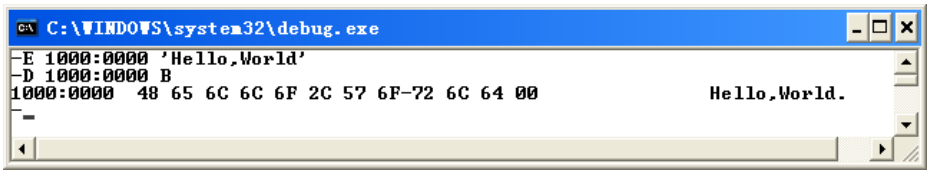


图 3.13　使用 E 命令向内存中写入字符串

3. 使用 R 命令查看、修改 CPU 寄存器的内容

R 是 Register 的头一个字母，R 命令用于查看、改变 CPU 寄存器的内容。

格式：

R

或

R [寄存器名]

如果要查看 AX、BX、CX、DX、CS、IP、SP、BP、SI、DI、DS、ES、SS 等寄存器和标志位的内容，可以在进入 Debug 后，在命令提示符“-”后输入“R”，按回车键，查询结果如图 3.14 所示。

注意到 CS 和 IP 的值分别为“0B4A”和“0100”，表示内存 0B4A：0100 处的指令为 CPU 将要读取执行的指令。在 Debug 窗口下方还列出了 CS:IP 所指向的内存单元中所存放的机器码，并且已将其翻译成汇编语言。

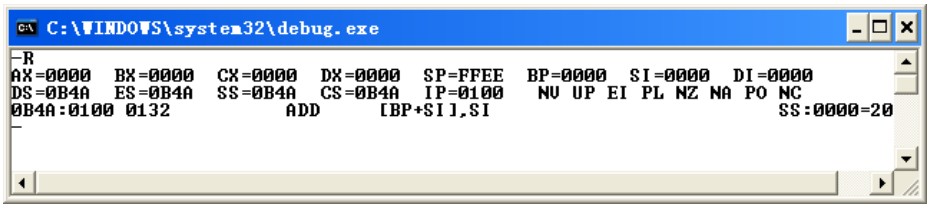


图 3.14 使用 R 命令查询寄存器内容

如果要修改一个寄存器中的值，如 AX 中的值，可以使用 R 命令加寄存器名来进行，如输入“R AX”后回车，将出现“:”作为输入提示，在后面输入要写入的数据后回车，即完成了对 AX 中内容的修改。若想查看修改的结果，可再用 R 命令查看，如图 3.15 所示，将 AX 寄存器值修改为“88”。

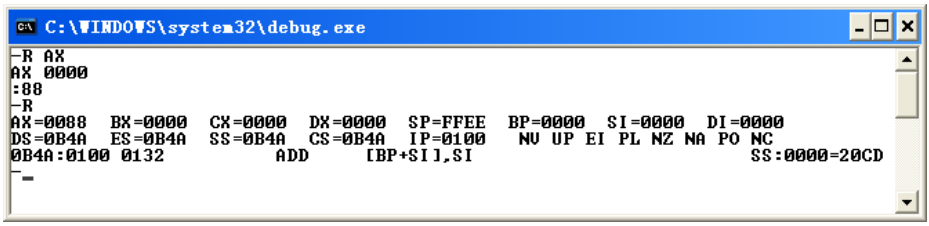


图 3.15 使用 R 命令修改寄存器内容

4. 使用 A 命令以汇编指令的格式在内存中写入机器指令

A 是 Assemble 的头一个字母，A 命令用于以汇编指令的格式在内存中写入机器指令。

格式：

A [段寄存器名]：[偏移地址]

或

A [段地址]:[偏移地址]

或

A [偏移地址]

或

A

当输入 A 命令后，显示段地址和偏移地址，等待用户输入汇编指令。每输入一条汇编指令回车后，自动显示下一条指令的段地址和偏移地址。再输入下一条汇编指令，直到汇

编语言程序全部输入。当又显示下一地址时，可直接回车，返回到提示符“-”为止。

如使用 A 命令从 1000:0 开始的内存单元中写入几条汇编语言，如图 3.16 所示。

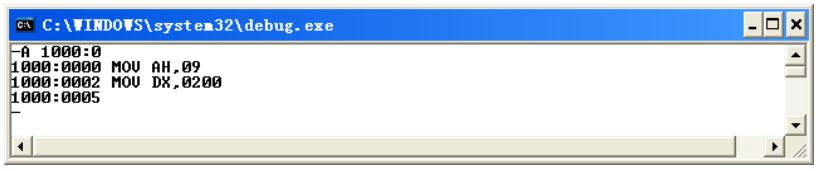


图 3.16　使用 A 命令写入汇编语言

5. 使用 T 命令跟踪与显示

T 是 Trace 的头一个字母，T 命令功能是跟踪与显示各个寄存器的内容。

格式：

T　[=地址]或 T[地址]

或

T　[=地址][条数]或 T[地址][条数]

其中第一种格式的功能是执行一条指定地址处的指令，停下来，显示 CPU 所有寄存器内容和全部标志位的状态，以及下一条指令的地址和内容。

第二种格式为多条跟踪命令，从指定的地址开始，若命令中用[地址]给定了起始地址，则从起始开始，若未给定，则从当前地址（CS：IP）开始。执行命令中的条数决定一共跟踪几条指令后返回 Debug 状态。

如使用 T 命令单条执行图 3.16 所示的两条指令，执行的结果如图 3.17 所示。

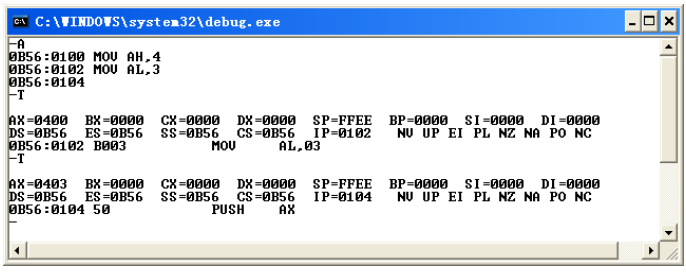


图 3.17　使用 T 命令跟踪和显示

6. 使用 U 命令将内存中的机器指令翻译成汇编指令

U 是 Unassemble 的头一个字母，U 命令将指定范围内的代码以汇编语言形式显示，同时显示该代码位于内存的地址。

格式：

U　[地址]

或

U　[地址范围]

例如可以使用 U 命令将从 1000:0 开始的内存单元中的内容翻译为汇编指令，并显示出来，如图 3.18 所示。

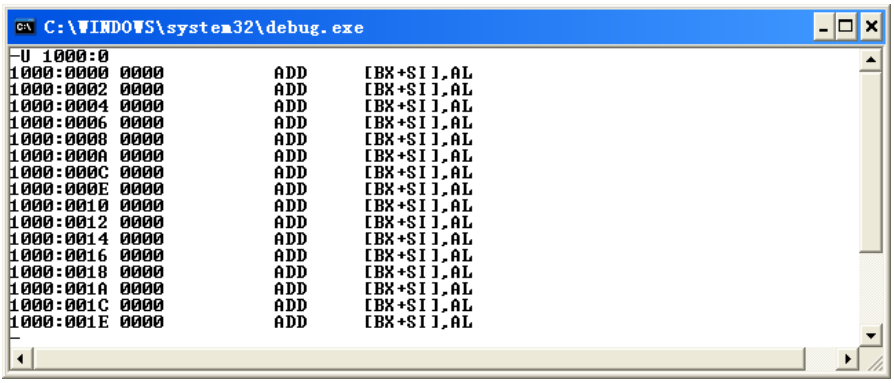


图 3.18　使用 U 命令将机器指令翻译成汇编指令

3.2　汇编语言程序

最早的程序设计都是采用机器语言来编写的。程序设计者必须将问题细化到每一条指令，其难度可想而知。而且程序员还必须牢记二进制数字组成的指令代码。程序员不仅要驾驭程序设计的全局，而且还要深入到每一个局部，所以即使是智力超群的程序员也常常效率低下。

ASCII 码的出现为改变机器指令的可读性和方便记忆提供了可能。既然 ASCII 码可以表示字符，字符又可以组成单词和短语，于是用助记符表示机器语言指令的汇编语言便产生了。

3.2.1　汇编语言

汇编语言和机器语言都是面向机器的语言，汇编语言是机器语言的符号化表示。汇编语言的出现为机器语言向高级语言的靠拢迈出了一大步，但是汇编语言还不是高级语言，它同机器语言一样都是低级语言。汇编语言程序需要经过汇编程序翻译成机器语言后才能被计算机执行。

汇编语言是汇编指令集、伪指令及其他使用规则的统称。其中，汇编指令是一些助记符，它们能够一对一地翻译成机器指令，伪指令是不对应具体内容的机器指令，它是程序设计者与汇编过程中的翻译程序、连接程序，以及执行时如何装入的一种约定。

在计算机中用汇编语言编写的程序称为汇编语言程序。汇编语言程序是一个文本文件，它必须经过翻译才能变成计算机可以执行的机器语言程序，在汇编语言程序设计中，这个翻译过程称为汇编。

3.2.2　汇编语言程序

汇编语言程序是由汇编语言编写的程序，汇编语言主要由汇编指令、汇编伪指令及其它元素构成。为了使读者对汇编程序有一个直观的了解，下面通过一个“Hello world”实

例来介绍汇编程序的书写样式。实例的主要功能同样是在屏幕上输出“Hello world”字符串，实例的主要代码如下：

```
data        SEGMENT
msg         DB   'Hello world $'
data        ENDS
code        SEGMENT
            ASSUME   CS: code, DS: data
start:
            MOV      AX , data
            MOV      DS , AX
            MOV      DX, OFFSET   msg
            MOV      AH, 9
            INT      21H
            MOV      AH, 4CH
            INT      21H
code        ENDS
            END      start
```

该汇编语言程序经过编译、连接和执行后，会在屏幕上输出字符串“Hello world”。在此仅是使读者对汇编程序设计有一个宏观上的认识，关于每一条语句的功能，将会在随后的章节中详细介绍。

从上面实例可以看出，实例可以分为两个独立的部分。第一部分从 data SEGMENT 开始到 data ENDS 结束，第二部分从 code SEGMENT 开始到 END start 结束。在汇编语言中，这两部分分别称为数据段和代码段，除此以外，在汇编语言程序设计中，可以附加数据段和堆栈段。在这四个段中，数据段和附加数据段主要的功能是定义在程序中使用的数据；堆栈段的功能是为程序的运行开辟堆栈空间；代码段是各种指令的结合，它的功能是完成逻辑功能。

汇编语言程序是一种严格的段结构程序。在汇编程序设计中，有三种段结构可供选择，分别是完成段结构、简化段结构和.com 程序结构。下面分别介绍这三种段结构。

1. 完整段结构

完整段结构应用非常多，它常用来开发一些规模比较大的汇编语言程序，完整段汇编源程序书写方式如下：

```
DATAS     SEGMENT
; 此处输入数据段伪指令
DATAS     ENDS
STACKS    SEGMENT
; 此处输入堆栈段伪指令
STACKS    ENDS
CODES     SEGMENT
          ASSUME  CS:CODES, DS:DATAS, SS:STACKS
START:
          MOV     AX, DATAS
          MOV     DS, AX
; 此处输入代码
```

```
        MOV     AH,4CH
        INT     21H
CODES   ENDS
END     START
```

2. 简化段结构

简化段结构汇编语言程序格式简单，易于掌握。简化段定义格式的汇编语言源程序的格式如下：

```
.model      small
.DATA
; 此处输入数据段伪指令
.STACKS
; 此处输入堆栈段伪指令
.CODES
START:
        MOV     AX, @DATA
        MOV     DS, AX
; 此处输入代码段代码
        MOV     AH, 4CH
        INT     21H
END     START
```

3. .com 程序结构

.com 程序又称为小规模程序，在.com 程序结构中，指令所需的数据直接放在代码段中。.com 程序结构定义格式的汇编语言源程序的格式如下：

```
.model      tiny
.code
ORG         100h
; 在此添加程序代码
        MOV     AH, 4CH
        INT     21H
END
```



注意：虽然汇编语言源程序对于大小写不敏感，但实际应用中应该尽可能统一。

3.3.3 Emu8086 的使用

1. Emu8086 的简介

Emu8086—Microprocessor Emulator 结合了一个先进的原始编辑器、组译器、反组译器、具有除错功能的软件模拟工具（虚拟 PC），还有一个循序渐进的指导工具。它可以在虚拟 PC 上一步一步地编辑、调试汇编语言程序，支持单步执行，而且机器指令的执行情况完全可视化，可以在程序执行的同时观察寄存器、标志位和内存。

模拟器会在虚拟 PC 中执行程序，这样可以隔绝程序，避免存取实际硬体，而在虚拟机器上执行组合程序，这可以让除错变得更加容易。同时它完全相容于 Intel 的下一代处理

器，包括了 Pentium Ⅱ、Pentium 4，这使得 8086 程序码的可携性相当高。

Emu8086 包含了学习汇编语言的全部内容。Emu8086 集源代码编辑器、汇编、反汇编工具，以及可以运行 debug 的模拟器（虚拟机器）于一身，此外，还有循序渐进的教程。这套软件对于刚开始学习汇编语言的用户非常有帮助，它能够编译源代码，并在模拟器上一步一步地执行。可视化界面操作可以在执行程序的同时观察寄存器、标志位和内存、算术和逻辑运算单元（ALU）显示中央处理器内部的工作情况。

8086 指令的一个优点是它的指令集非常小，学起来会容易得多。Emu8086 同主流汇编程序相比，语法简单得多，但是它能生成任何能兼容 8086 机器语言的代码。

注意：如果不使用 Emu8086 编译程序，则无法在运行的时候单步跟踪。

Emu8086 官方下载网址：http://www.emu8086.com/。

2. Emu8086 的安装

在下载完 Emu8086 应用软件的安装文件“emu8086v408r.exe”后，直接双击此文件启动安装向导，如图 3.19 所示，接下来的过程都是直接单击“Next”按钮即可完成安装。安装完成后启动 Emu8086 的效果如图 3.20 所示。

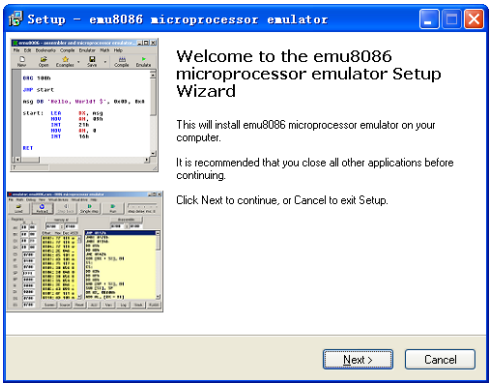


图 3.19　安装 Emu8086

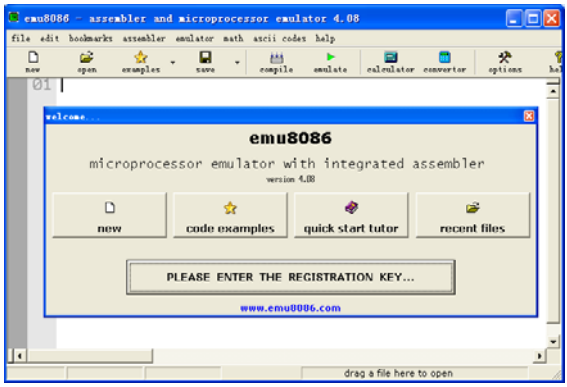


图 3.20　启动后的 Emu8086

3. Emu8086 的使用

Emu8086 可以在 Windows 平台下运行，各种调试窗口为调试汇编语言程序提供了极大的方便，同时也还可以模拟 DOS 的图形显示环境，为学习、理解和设计图形化程序提供了参考。

【例 3.3】　交通灯仿真。

这一练习是模拟交通灯进行交通仿真。

步骤如下：

（1）运行 Emu8086 模拟器。依次选择【开始】→【所有程序】→【emu8086】→【emu8086】，打开 Emu8086 模拟器，如图 3.20 所示。

（2）新建文件。在“welcome”对话框中单击“new”按钮，弹出“choose code template”对话框，在此对话框中选中“emply workspace”单选框，如图 3.21 所示，单击“OK”按钮。

（3）输入汇编程序代码。在仿真器中输入汇编程序代码，代码如下：

```
#start=Traffic_Lights.exe#
NAME            "traffic"
```

```
        MOV     AX, all_red
        OUT     4, AX
        MOV     SI, offset situation
NEXT:
        MOV     AX, [SI]
        OUT     4, AX

        ; 等 5 秒 (5 million microseconds)
        MOV     CX, 4CH         ; 004C4B40H = 5,000,000
        MOV     DX, 4B40H
        MOV     AH, 86H
        INT     15H

        ADD     SI, 2           ; 下个位置
        CMP     SI, sit_end
        JB      NEXT
        MOV     SI, offset situation
        JMP     NEXT

;       FEDC_BA98_7654_3210
situation:
s1      DW      0000_0011_0000_1100B
s2      DW      0000_0110_1001_1010B
s3      DW      0000_1000_0110_0001B
s4      DW      0000_1000_0110_0001B
s5      DW      0000_0100_1101_0011B
sit_end = $

all_red EQU     0000_0010_0100_1001B
```

（4）运行程序。在仿真器中单击emulate图标按钮，弹出“original source code”、“emulator”和“Traffic Lights”窗口，在“emulator”窗口中单击“run”按钮运行程序，“emulator”和“Traffic Lights”窗口分别如图 3.21 和 3.22 所示。

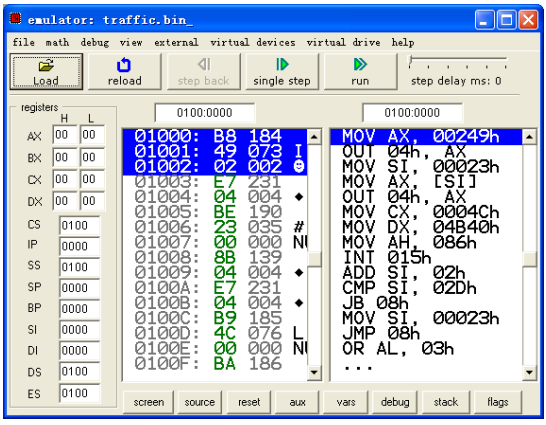


图 3.21　emulator 窗口

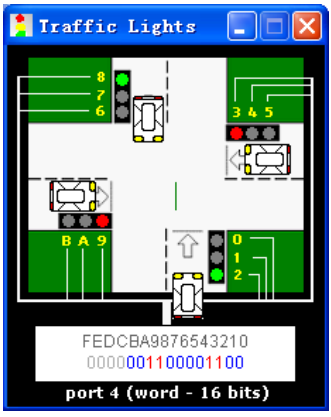


图 3.22　Traffic Lights 窗口

程序分析：

源程序中第一句#start=Traffic_Lights.exe#是关键，它启动了一个可执行程序Traffic_Lights.exe，用来模拟交通灯。

3.3　汇编语言程序的编辑、编译、连接和运行

汇编语言程序和机器语言程序虽然都是面向机器的程序设计语言，但是计算机不能识别汇编语言指令，汇编语言源程序必须经过编译和连接后才能被计算机执行。对于程序员来说，开发汇编语言程序，需要经过编辑、编译、连接和运行等过程，如图 3.23 所示。



图 3.23　汇编语言程序开发过程

3.3.1　汇编语言程序的编辑

汇编语言源程序是一个文本文件，其扩展名为.asm，因此汇编语言源程序文件可以由任何文本编辑工具创建，包括 Windows 下的“记事本”、“写字板”，甚至 Word 也可以，不过需要注意的是，如果使用 Word 编辑一定要保存为无格式的纯文本文件，而不是.doc 或其他文件格式。另外 Windows 平台都附带了“Edit”执行文件，汇编语言源程序的编辑也可以使用 Edit 完成。

除 Win32 汇编外，一般情况下汇编程序大都在命令行方式下运行。本书重点介绍如何利用 DOS 环境下的 Edit 编辑汇编语言源程序。

【例 3.4】　使用 Edit 编辑汇编语言程序，该程序的功能是键盘输入一个字母后，将字母在显示器上显示出来。

步骤如下：

1. 打开 Edit 编辑环境

在 Windows 操作系统下，依次选择【开始】→【运行】选项，在弹出的“运行”对话框中输入“edit”或“edit.com”，如图 3.24 所示。单击“确定”按钮，即可进入 DOS 系统环境下的 Edit 编辑环境，如图 3.25 所示。

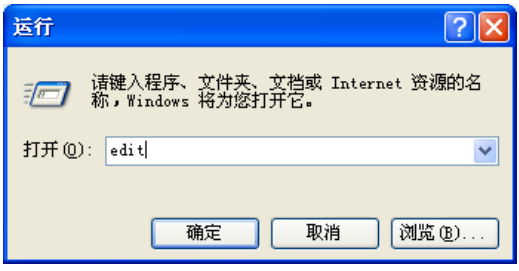


图 3.24　Windows 运行对话框

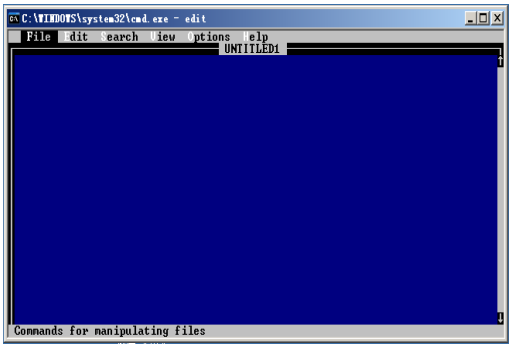


图 3.25　Edit 编辑器

2. 使用 Edit 编辑汇编语言源程序

Edit 是 DOS 环境下的一个编辑软件，打开 Edit 编辑窗口后，可以通过“Alt+Enter”组合键把 Edit 窗口切换到全屏窗口，在全屏窗口下 Edit 编辑器的样式跟纯 DOS 环境相似。如果想退出全屏模式，按“Alt+Enter”组合键即可。

Edit 编辑窗口的上方是菜单选项：File、Edit、Search、View、Option 和 Help。可以通过“Alt+菜单项的首字母”打开相应的菜单选项。例如按“Alt+F”组合键可以打开 File 菜单。在打开菜单的下级菜单中，可以看到菜单项中有一个白色的字母，同样可以通过“Alt+菜单项的白色字母”组合键即可打开相应的下级菜单。如在 Edit 编辑器中输入如下代码，其中“INT 21H”重复 3 次，可以首先选中此行代码，按“Ctrl+C”组合键复制此行代码，按“Ctrl+V”组合键粘贴此行代码，代码如下：

```
.MODEL   TINY
.CODE
ORG      100H
         MOV      AH, 1
         INT      21H
         MOV      DL, AL
         MOV      AH, 2
         INT      21H
         MOV      AH, 4CH
         INT      21H
END
```

因为 Edit 是 Windows 平台附带的可执行文件，以上所有的操作也可以通过鼠标来完成。关于鼠标的操作和在 Windows 下类似，在此不做赘述。

3. 保存汇编语言程序

编辑完代码后，按“Alt+F”组合键打开“File”菜单，在弹出的“File”子菜单中通过键盘组合键“Alt+S”选择“Save”命令，弹出“Save As”对话框。在“Save As”对话框中“File Name”为所要保存的文件名，“Directories”文件存放的目录，“Existing Files”为当前目录中所有的文件。

输入文件名“EX3-4.asm”，保存在“E”盘，具体设置如图 3.26 所示。然后单击“OK”按钮完成保存文件。

4. 退出 Edit

按“Alt+F”组合键打开“File”菜单，在弹出的“File”子菜单中通过键盘组合键“Alt+X”退出 Edit。

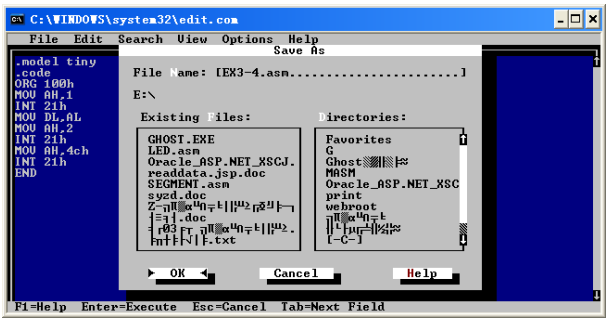


图 3.26　保存源程序

3.3.2　MASM 汇编开发环境

MASM 是微软公司开发的汇编开发环境，拥有可视化的开发界面，使开发人员不必再使用 DOS 环境进行汇编的开发，编译速度快，支持 80x86 汇编及 Win32Asm，是 Windows 下开发汇编的利器。它与 Windows 平台的磨合程度非常好，但是在其他平台上就有所限制，使用 MASM 的开人员必须在 Windows 下进行开发。

1. MASM 的安装

本书使用的是 MASM 6.11 版本，安装步骤如下：

（1）打开安装文件“MASM611”文件夹，双击“SETUP.EXE”文件运行安装程序。在出现的“Microsoft MASM 6.11 Setup”屏幕之后，直接按“Enter”键继续安装，如图 3.27 所示。

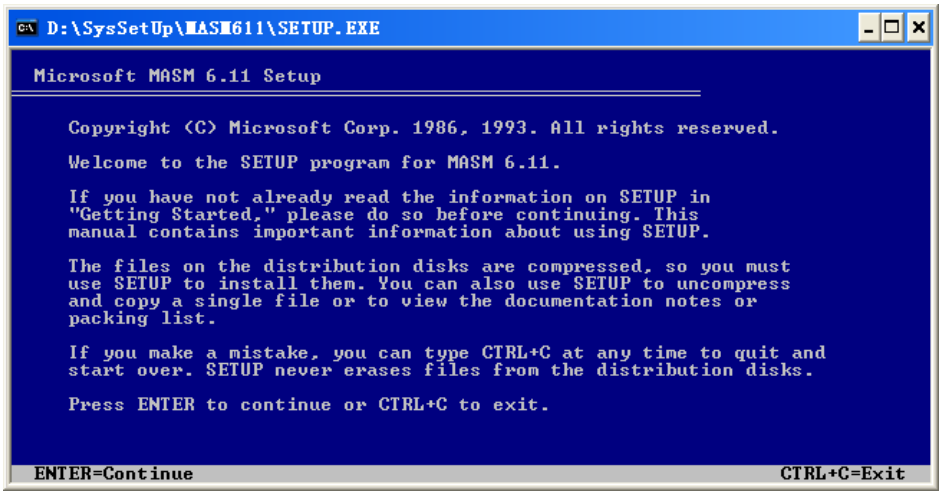


图 3.27　安装 MASM 首界面

（2）系统出现的提示信息及其用户进行的操作如表 3.1 所示。

表 3.1　MASM 安装出现的提示信息及其用户进行的操作

出现提示信息	根据情况进行操作
Microsoft MASM 6.11 Setup Main Menu	选中“Install the Microsoft Macro Assembler”
Microsoft MASM 6.11 Setup System Configuration	选中第 2 个“MS-DOS/Microsoft Windows”，如果选择第一个将无法 Debug
Microsoft MASM 6.11 Setup Microsoft Windows Files	对于“Install Files for Microsoft Windows？”，选中“Yes”
Microsoft MASM 6.11 Setup Programmer’s Workbench	对于“Install PWB？”，选中“Yes”
Microsoft MASM 6.11 Setup Brief Compatibility	对于“Install Brief Compatibility？”，选中“No”
Microsoft MASM 6.11 Setup Mouse Driver	对于“Copy the Microsoft Mouse Driver？”，选中“Yes”
Microsoft MASM 6.11 Setup MAS.EXE Utility”	对于“Install the MASM.EXE Utility？”，选中“Yes”
Microsoft MASM 6.11 Setup Help Files	对于“Copy the Help Files？”，选中“Yes”
Microsoft MASM 6.11 Setup Sample Programs	对于“Copy the Sample Programs？”，使用上下键选中“Yes”

续表

出现提示信息	根据情况进行操作
Microsoft MASM 6.11 Setup Destination Drive	对于“Please Select the Destination Drive:”中所列出的硬盘驱动器，可任选其一。此处选中“Hard Drive D”
Microsoft MASM 6.11 Setup Executable Files	对于“D:\ MASM611\BIN”，不改变
Microsoft MASM 6.11 Setup Library Files	对于“D:\ MASM611\LIB”，不改变
Microsoft MASM 6.11 Setup Include Files	对于“D:\ MASM611\INCLUDE”，不改变
Microsoft MASM 6.11 Setup Initialization Files	对于“D:\ MASM611\INIT”，不改变
Microsoft MASM 6.11 Setup Help Files	对于“D:\ MASM611\HELP”，不改变
Microsoft MASM 6.11 Setup Sample Files	对于“D:\ MASM611\SAMPLES”，不改变

（3）出现“Microsoft MASM 6.11 Setup Confirm Your Choices”屏幕，列出了前面各步骤所做出的各项选择的清单，供用户校核，如图 3.28 所示，如果发现有选择不当的项目，依然可以进行修改。

修改的方法是，先用箭头键选中欲修改的项目，按“Enter”键，然后再用箭头键选中“Yes”或“No”，最后再按“Enter”键继续。

如果校核无误，可以按“Enter”键继续，以给出确认信息。

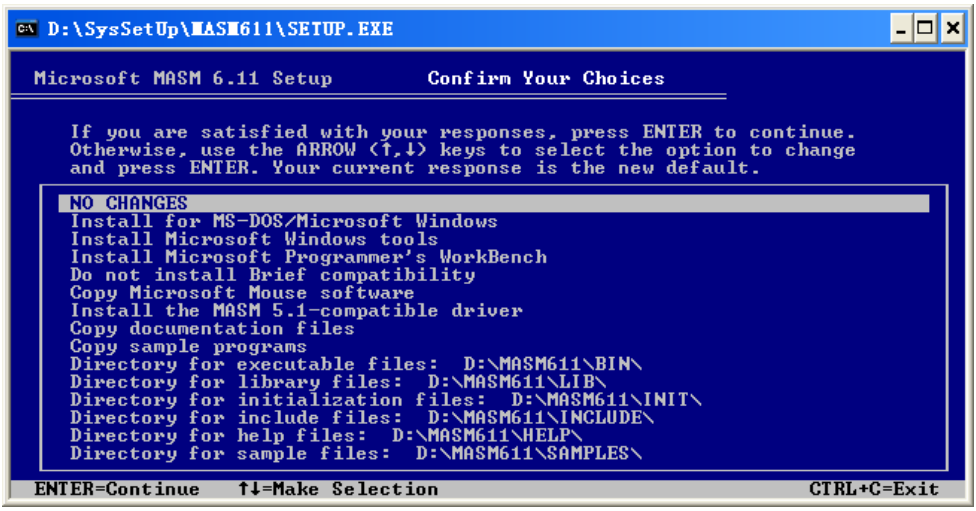


图 3.28　确认安装

（4）出现“Microsoft MASM 6.11 Setup Installing”屏幕，安装程序进行复制文件的操作，此时仅需等待一会儿。

（5）复制文件操作结束之后，出现“Microsoft MASM 6.11 Setup Installation Complete”屏幕，按“Ctrl+C”组合键，退出安装。

2. MASM 的启动

MASM 6.11 的人机操作接口有两类窗口：一类是 PWB（Programmer’s Work Bench）窗口，支持用户进行汇编语言源程序文件的编辑、汇编和连接操作。另一类是 CodeView 窗口，支持用户进行各种调试操作。

MASM 6.11 的启动过程是，首先启动 PWB 的主窗口，然后通过 PWB 的主窗口，再

启动 PWB 的其他子窗口和启动 CodeView 类窗口。

PWB 主窗口的启动方法如下：

（1）打开路径为“D:\ MASM611\INIT”的文件夹，把其中的文件“Tools.pre”改名为“Tools.ini”。

（2）打开路径为“D:\ MASM611\BIN”的文件夹，运行其中的文件“PWB.EXE”，屏幕上即会出现 PWB 的主窗口，如图 3.29 所示。

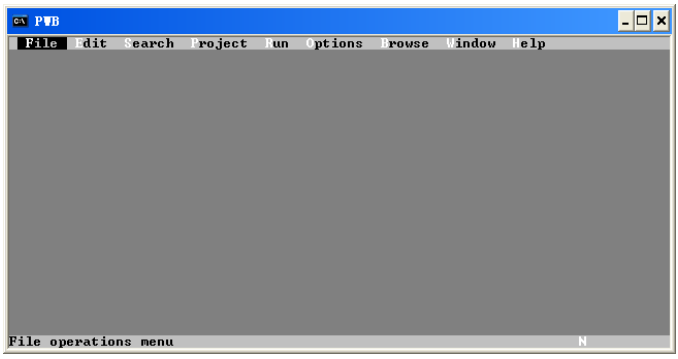


图 3.29　PWB 窗口

3. Editor Settings 项目的设置

在 PWB 的主窗口中，选择“Options”菜单项。在弹出的子菜单项中，选择“Editor Settings”子菜单，弹出“Editor Settings”窗口，如图 3.30 所示，在“Editor Settings”窗口中，应进行开关“autosave”、“envcursave”和“envprojsave”的修改设置，即把它们由“no”改为 “yes”，操作步骤如下：

（1）在“Switch List”项目下列出的开关清单上，单击欲修改设置的开关。

（2）在“Switch”项目下，进行开关的状态修改，每次一个开关。

（3）单击“Set Switch”按钮。

（4）重复步骤（1）、（2）、（3），直至把欲修改设置的开关全部完成为止。

（5）单击“Save”按钮。

（6）在弹出的“Save Editor Settings”窗口中，检查“Modified Switches List”项目下所列出的开关清单应正确无误，之后单击“OK”按钮。

（7）单击“Editor Settings”窗口中的“OK”按钮完成设置。

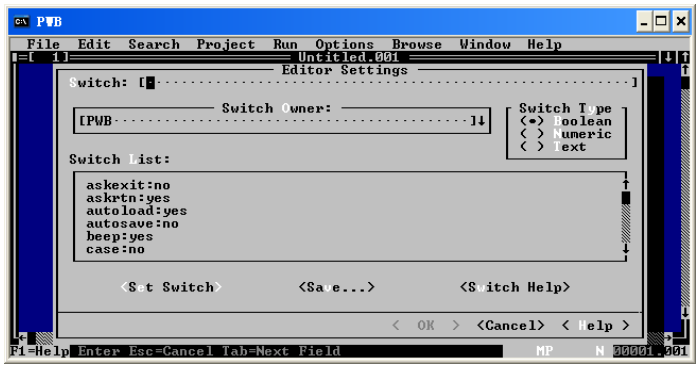


图 3.30　Editor Settings 的设置

4. Environment Variables 项目的设置

在 PWB 的主窗口中，选择“Options”菜单项，在弹出的子菜单项中，选择“Environment Variables”子菜单，弹出“Environment Variables”窗口，在“Environment Variables”窗口中，在原有的“PATH=”行上增加“D:\mymasm”和“D:\MASM611\BIN”两项。

其中“D:\mymasm”是用户在启动 PWB 之前自建的工作目录。增加如下项目：

LIB=D:\MASM611\LIB，

INCLUDE=D:\MASM611\INCLUDE，

INIT=D:\MASM611\INIT

HELPFILES=D:\MASM611\HELP*.HLP，

ASMEX=D:\MASM611\SAMPLES。

修改和增加项目的步骤如下（如图 3.31 所示）：

（1）在“Variable”项目下输入欲修改或增加的项目。

（2）单击“Set Variable”按钮。

（3）重复步骤（1）、（2），直至欲修改和增加的项目全部完成为止。

（4）单击“OK”按钮。

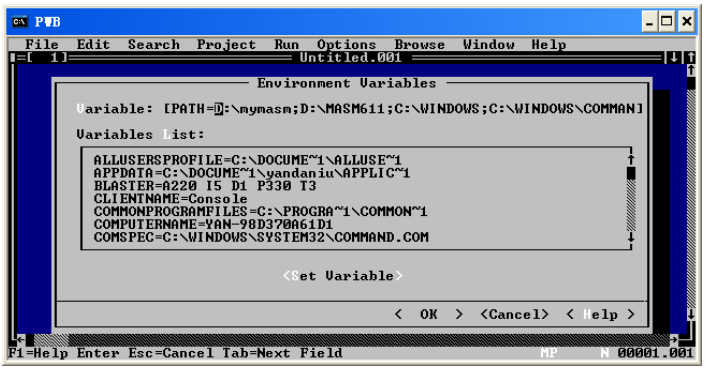


图 3.31　Environment Variables 的设置

5. Build Options 项目的设置

在 PWB 的主窗口中，选择“Options”菜单，在弹出的子菜单中，选择“Build Options”子菜单，弹出“Build Options”窗口，在“Build Options ”窗口中，选中“Use Debug Options”。单击“OK”按钮完成 Build Options 的设置，如图 3.32 所示。

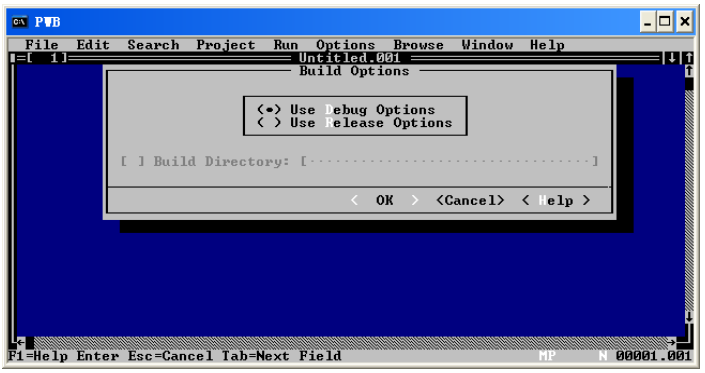


图 3.32　Build Options 的设置

6. MASM 安装后的文件结构

安装完 MASM 后，在 D:\MASM611 目录下，其结构如图 3.33 所示。MASM 6.11 提供的编译、连接等工具主要存放在 BIN 文件夹下，在该文件下有四个最重要的文件，分别是 MASM.EXE、LINK.EXE、ML.EXE 和 PWB.EXE。其中前三个是 Microsoft 提供的编译连接工具，最后一个是一个源文件编辑工具，具体使用方法同 Edit。

MASM 6.11 的开发编译程序为 MASM.EXE，连接程序为 LINK.EXE。从 MASM 6.0 版本开始，Microsoft 将源程序的编译和连接合到一个单一程序中，并命名为 ML.EXE，因此当准备好源程序后，即可以分别使用 MASM.EXE 和 LINK.EXE 程序进行编译和连接，也可以直接使用 ML 进行编译和连接。

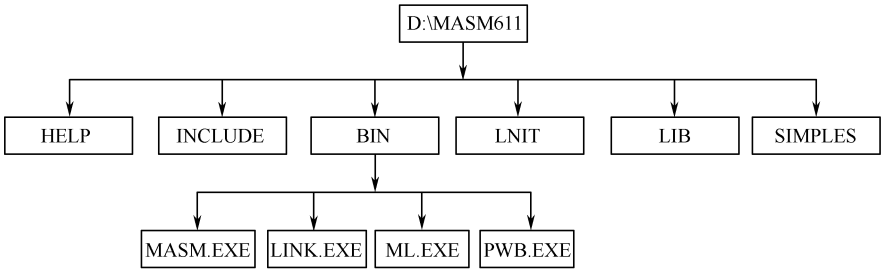


图 3.33　MASM 6.11 目录结构

3.3.3　汇编语言程序的编译、连接和运行

编写汇编语言程序至少需要三个工具，它们分别是用来编辑源文件的编辑程序、用来将.asm 程序转化成目标文件的 MASM 汇编程序，以及用来将多个目标程序连接成一个可执行文件的 Link 连接程序。在 3.3.1 节中，已经介绍了汇编程序的编辑，在 3.3.2 节中，介绍了 MASM。本节主要介绍使用 MASM 对汇编语言程序的编译、连接和运行。

1. 源程序的编辑

可以使用任意文本编辑器来编辑汇编语言程序源文件，只要最终将其存储为纯文本文件即可，并且文件的扩展名是.asm，这里使用前面介绍的 DOS 下的 Edit 来编辑。汇编语言源程序不能被计算机直接执行，必须通过编译工具编译、连接后才能被计算机执行。

在这里以输出笑脸的汇编语言程序为例，介绍汇编语言程序的编译、连接和运行。使用 Edit 编辑源程序，在前面介绍过，这里就不做介绍，保存的文件名为“face.asm”。

代码如下：

```
data    SEGMENT
        DB      'a'                    ; 在数据段中定义了存储了两个数据
        DB      'b'
data    ENDS
code    SEGMENT
        ASSUME  CS:code , DS:data
start:
        MOV     AX , data
```

```
        MOV     DS , AX
        MOV     DL , [0]                ; 内存直接寻址，相对于 DS 段，EA=0
        MOV     AH , 2
        INT     21H                     ; 调用 DOS 系统 2 号功能，输出数据
        MOV     DL , [1]                ; 内存直接寻址，相对于 DS 段，EA=1
        INT     21H
        MOV     AH , 4CH
        INT     21H                     ; 调用 DOS 系统 4CH 号功能，返回 DOS 系统
code    ENDS
        END     start
```

2. 源程序的编译

编译是将源程序转化成目标文件的过程。假设在 D:\test 目录下有需要编译源程序，则使用 MASM 6.11 编译汇编源程序文件的格式如下：

格式：

```
D:\test>MASM SORURCE<.asm>,[OUT <.obj>],[list <.list>],[cref <.cref>]
```

其中，SOURCE 是必需的，它指明了需要编译的源程序，[OUT <obj>],[list <.list>],[cref <.cref>]三项是可选的。[OUT <obj>]项指明了编译程序生成的目标文件名，[list <.list>]项用于产生列表文件，列表文件保存的是编译过程中产生的中间结果。[cref <.cref>]项用于产生交叉引用文件，交叉引用文件同列表引用文件相同，它也保存由源程序转化为目标程序的中间结果，各个选项之间用“,”分开。

编译源文件 face.asm 的步骤如下：

（1）复制文件。在 D 盘目录下新建一个文件夹，命名为“test”，将 D:\MASM6.11\BIN 目录下的“MASM.EXE”和“LINK.EXE”文件复制到 test 文件夹中。

（2）进入 DOS 环境。在 Windows 操作系统下，依次选择【开始】→【运行】选项，在弹出的“运行”对话框中输入“cmd”，单击“确定”按钮，即可进入 Windows 环境下的 DOS 窗口。

（3）进入 test 目录。为了方便后继的调试，在 DOS 下输入“D:”后按 Enter 键，设置 D 盘为当前盘。再输入“CD\test”后按回车键，将 test 设置为当前目录。

（4）编译源程序。在 DOS 下输入“MASM face.asm”命令后按 Enter 键，在“D:\test”目录下生成了“face.obj”目标文件。

整个操作步骤如图 3.34 所示。

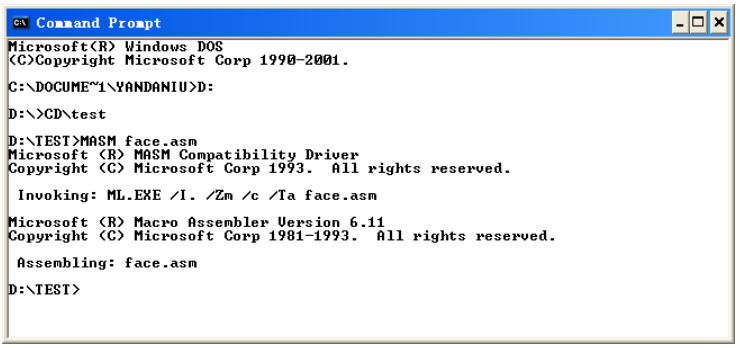


图 3.34　编译源文件

3. 连接目标文件

当源程序被编译成目标文件后，需要对目标文件进行连接后才能生成可执行文件。连接目标文件采用 LINK 命令，格式如下：

格式：

```
D:\test>LINK OBJ1<.obj>,[OBJ2 <.obj>]…
```

其中，OBJ1 和 OBJ2 为要连接的目标文件，输入需要连接的目标文件，如果有多个文件，则在各个文件之间用“,”隔开。

连接目标文件 face.obj 的步骤如下：

（1）连接目标文件。在 DOS 下输入“LINK face.obj”命令后按 Enter 键，提示输入生成的可执行文件名称（Run Fil[face.exe]:），如图 3.35 所示，如果想采用默认的文件名，直接按 Enter 键即可。在这里使用默认的文件名，按 Enter 键继续。

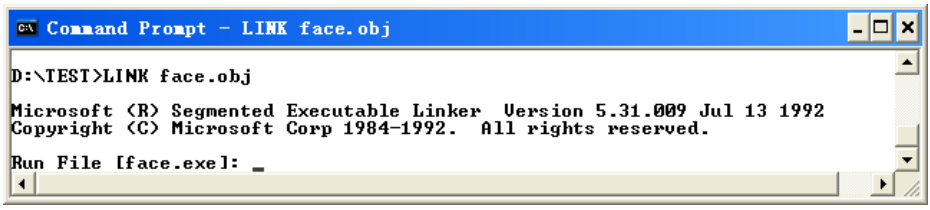


图 3.35　连接目标文件

（2）输入映像文件名称。继上一步后系统提示输入映像文件名称（List File[nul.map]:），映像文件用来保存目标文件转换成可执行文件的中间结果。如果不想生成这个文件，直接按 Enter 键，在这里直接按 Enter 键。

（3）输入库文件的名称。继上一步后系统提示输入库文件的名称（Libraries[.lib]:），库文件包含了一些可以调用的子程序，如果程序中调用了某一个库文件的子程序，就需要在连接的时候将这个库文件和目标文件连接到一起。如果没有引用库文件中的子程序，可以忽略此项，直接按 Enter 键即可，在这里直接按 Enter 键。

（4）输入定义文件名称。继上一步后系统提示输入定义文件名称（Definitions File[nul.def]:），此项一般可以忽略，直接按 Enter 键即可。

整个过程如图 3.36 所示，在 D:\test 目录下生成了“FACE.EXE”文件。

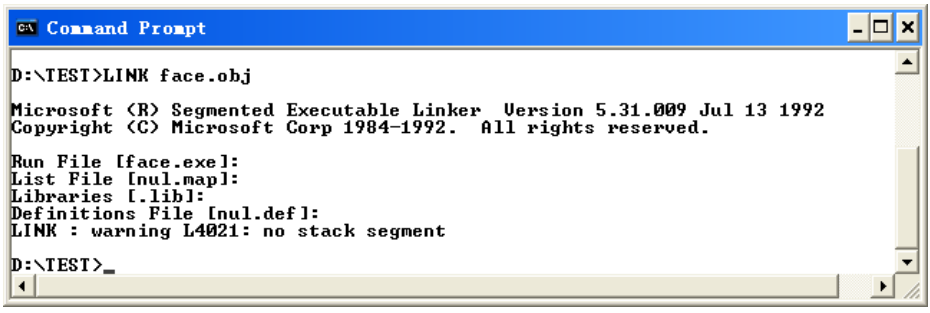


图 3.36　连接目标文件过程

4. 运行

运行程序比较简单，直接在 DOS 下运行生成的可执行文件。格式如下：

格式：

```
D:\test>File<.exe>
```

其中，File 为可执行文件。

运行 FACE.EXE 的方法是在 DOS 下输入“FACE.EXE”，按 Enter 键运行此程序，结果如图 3.37 所示。

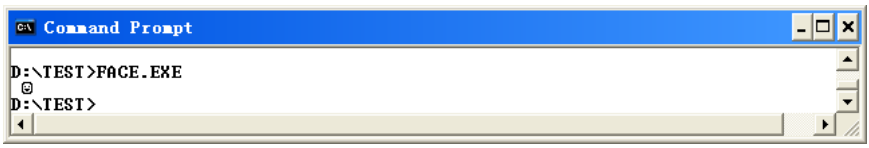


图 3.37　程序的运行

说明：

在汇编语言程序设计中，LINK 程序主要应用在以下两种场合。

（1）源程序很大，由多个源程序文件组成，每一个源程序编译成目标代码后，需要 LINK 程序将各个目标模块连接在一起，形成可执行文件。

（2）程序中调用了某一个库的子程序时，需要 LINK 程序将该库文件和目标模块连接在一起。

除了这两种场合以外，对于一些源程序文件，可直接采用 ML 程序进行编译、连接。其使用格式如下：

格式：

```
D:\test>ML SORURCE <.asm>
```

其中，SOURCE 是必需的，它指明了需要编译连接的源程序。

如使用 ML 程序对 face.asm 源文件进行编译，连接的步骤如下：

（1）复制文件。将 D:\MASM6.11\BIN 目录下的“ML.EXE”文件复制到 test 文件夹中。

（2）进入 DOS 环境。在 Windows 操作系统下，依次选择【开始】→【运行】选项，在弹出的“运行”对话框中输入“cmd”，单击“确定”按钮，即可进入 Windows 环境下的 DOS 窗口。

（3）进入 test 目录。为了方便后继的调试，在 DOS 下输入“D:”后按 Enter 键，设置 D 盘为当前盘。再输入“CD\test”后按回车键，将 test 设置为当前目录。

（4）编译连接源程序。在 DOS 下输入“ML face.asm”，按 Enter 键编译连接源程序，在 D:\test 目录下生成了“FACE.EXE”和“face.obj”文件。

（5）运行 FACE.EXE 的方法是在 DOS 下输入“FACE.EXE”，按 Enter 键运行此程序，结果如图 3.37 所示。

整个操作过程如图 3.38 所示。

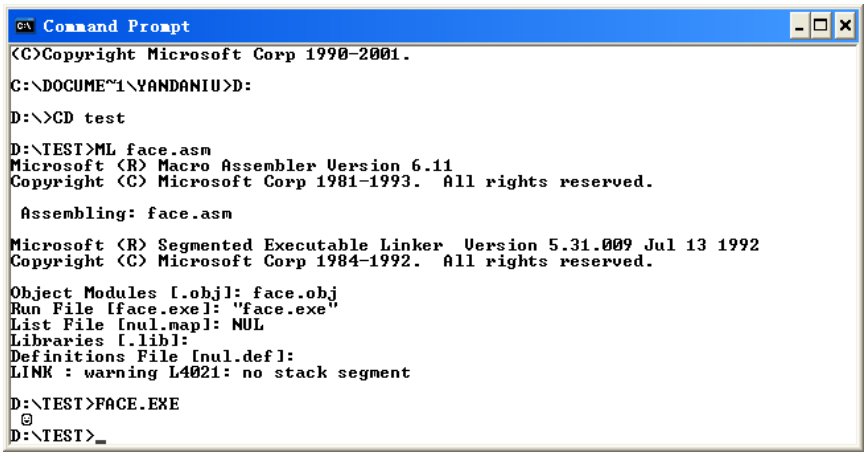


图 3.38　使用 ML 程序编译连接的操作过程

习题

1．机器语言程序的主要特点有哪些？

2．汇编语言的主要特点有哪些？

3．汇编语言有哪几种段结构？

4．Debug 练习。

（1）参考本章的例 2.1 和例 2.2，在 Debug 下调试汇编程序，并输出“Assembly”。

（2）练习使用常用的 Debug 命令。

5．Emu8086 练习。

（1）试验一个显示“I Love Assembly”的程序。

```
data    SEGMENT
msg     DB      ' I Love Assembly $'
data    ENDS
code    SEGMENT
        ASSUME  CS: code, DS: data
start:
        MOV     AX , data
        MOV     DS , AX
        MOV     DX, OFFSET  msg
        MOV     AH, 9
        INT     21H
        MOV     AH, 4CH
        INT     21H
code    ENDS
        END     start
```

（2）在 Emu8086 中调试运行上面的程序，并单步执行该程序，记录下每执行一条指令后相应寄存器内容的变化情况。

（3）解释各个窗口界面的作用和意义。

6．MASM 练习

（1）使用 Edit 编辑如下源代码。

```
DATA    SEGMENT
        DB      'C'                          ; 在数据段中定义了存储了两个数据
        DB      'D'
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE , DS:DATA
START:
        MOV     AX , DATA
        MOV     DS , AX
        MOV     DL , [0]            ; 内存直接寻址，相对于 DS 段，EA=0
        MOV     AH , 2
        INT     21H                 ; 调用 DOS 系统 2 号功能，输出数据
        MOV     DL , [1]            ; 内存直接寻址，相对于 DS 段，EA=1
        INT     21H
        MOV     AH , 4CH
        INT     21H                 ; 调用 DOS 系统 4CH 号功能，返回 DOS 系统
CODE    ENDS
        END     START
```

（2）使用 MASM 和 LINK 命令行方式命令建立可执行文件。

（3）使用 Debug 进行调试和运行。

（4）观察并记录实验数据和实验结果。

第 4 章

汇编语言指令系统和寻址方式

计算机是通过执行指令序列来解决问题的，计算机的指令是由操作码和操作数组成的。操作数如何获得和存储就是指令的寻址方式。根据存放位置的不同，汇编指令中的操作数可以分为立即操作数、寄存器操作数、存储器操作数和 I/O 操作数。与此相对应便有四种寻址方式：立即寻址、寄存器寻址、存储器寻址和 I/O 端口寻址。

本章主要介绍汇编语言指令格式、四种类型的操作数，以及相对应的寻址方式，同时对指令系统做简单介绍，在后面章节中将详细介绍这些指令系统。

4.1 汇编语言指令的格式

指令是计算机用以控制各个部件协调工作的命令，微型计算机中的指令系统是指微处理器所能执行的各种指令的集合，微处理器的主要功能是通过它的指令系统实现的。

指令由操作码和操作数两部分组成。操作码说明计算机要执行哪种操作，如传送、运算、移位、跳转等，它是指令中不可缺少的组成部分。操作数是指令执行的参与者，也就是各种操作的对象。有些指令不需要操作数，通常的指令都有一个或两个操作数，也有个别指令有 3 个甚至 4 个操作数。指令格式通常如下：

格式：

```
<标号：>操作码  <操作数 1> <，操作数 2>      <；注释>
```

其中，

（1）操作码：又称为指令助记符，指示计算机所要执行的操作类型，如加法、减法、数据传送等。

（2）“<>”号：其中内容可有可无。

（3）标号：必须是字母打头的由字母或数字组成的字符串。标号供转移指令作为转移的目标。一般情况下都没有标号，如果有标号，则其在程序单位中必须是唯一的。

（4）操作数：在双操作数的指令中，操作数 2 称为源操作数，表示参与指令操作的一个对象；操作数 1 称为目的操作数，它不仅可以作为指令操作的一个对象，还可以用来存放指令操作的结果。

（5）“;”号：表示其后面的内容为注释部分，由“;”起直至<ENTER>。汇编语言程序在其译成机器指令时将无视这一部分，而仅在输出源程序清单时才原样输出。在输入源程序时，每一条汇编指令的末尾必须输入换行键<ENTER>，<ENTER>表示本指令的结束，下一指令的开始。

【例 4.1】 编写带有标号、双操作数、单操作数和固定操作数指令的汇编语言。

代码如下：

```
L1:  MOV     AX,10H   ; 带标号，双操作数指令
     INC     21H      ; 单操作数指令
     AAA              ; 固定操作数
```

程序分析：

（1）第一条汇编指令前的 L1 是一标号，MOV 是操作码，AX 是目标操作数，指出要将结果存入寄存器 AX 中，10 是源操作数，指出指令处理的数来自何处，这里就是十六进制数 10H 本身，该指令执行结果就是将 10H 送入 AX 中。“;”后是注释部分，在 DOS 下运行汇编程序，注释部分只能用英文和 ASCII 码符号书写，在中文操作系统下则可用中文书写。

（2）第二条指令中，INC 是操作码，21H 是目标操作数。指令功能是将 21H 寄存器的内容加 1 后将结果送入 21H。

（3）第三条指令中，AAA 是操作码，这一指令隐含使用操作数 AL。

由以上 3 条指令可以看出，这里所说的操作码通常是指指令功能的助记符，它给出了指令功能便于记忆的形式，例如，MOV 是 Movement 的缩写等。

在后面章节中，将为读者介绍每个操作码的功能。而操作数的给出方式则比较复杂。操作数可以是一个具体的数值，也可以是存放数据的寄存器或存储器地址，把寻找操作数的方式叫做操作数的寻址方式。操作数采取哪一种寻址方式，会影响机器运行的速度和效率。而如何寻址一个操作数，对程序的设计来讲也是很重要的。后面将详细介绍 Intel 8086 的寻址方式，这是掌握指令功能的关键。

4.2 指令系统分类

Intel 8086/8088 CPU 的指令可以是单字节指令，也可以是多字节指令。从功能上可以分成 6 类，即数据传送类指令、算术运算类指令、位操作类指令、控制转移类指令、字符串处理类指令和处理器控制类指令。本节只是简单介绍各种指令，在后续的章节中会详细介绍各种指令的格式及功能。

4.2.1 数据传送类指令

数据传送类指令包括 4 组指令：通用数据传送指令、输入/输出指令、地址目标传送指令和标志位传送指令。

1. 通用数据传送指令

（1）MOV（MOVe byte or word）：传送字节或字。

（2）POP（POP word off stack）：字退栈。
（3）PUSH（PUSH word into stack）：字进栈。
（4）XCHG（eXCHanGe byte or word）：字节或字交换。
（5）XLAT（transLATe byte）：字节换码。

2. 输入/输出指令

（1）IN（INput byte or word）：输入字节或字。
（2）OUT（OUTput byte or word）：输出字节或字。

3. 地址目标传送指令

（1）LEA（Load Effective Address）：取有效地址。
（2）LDS（Load pointer using DS）：取指示器到 DS。
（3）LES（Load pointer using ES）：取指示器到 ES。

4. 标志位传送指令

（1）LAHF（Load AH register from Flags）：取标志到 AH。
（2）SAHF（Store AH register in Flags）：取 AH 到标志。
（3）PUSHF（PUSH Flags into stack）：标志进栈。
（4）POPF（POP Flags off stack）：标志退栈。

4.2.2　算术运算类指令

算术运算类指令包括 4 组指令：加法指令、减法指令、乘法指令和除法指令。

1. 加法指令

（1）ADD（ADD byte or word）：字节或字相加。
（2）ADC（ADd byte or word with Carry）：带进位的字节或字相加。
（3）INC（INCrement byte or word by l）：字节或字加 1。
（4）AAA（ASCII Adjust for Addition）：加法的 ASCII 码修正。
（5）DAA（Decimal Adjust for Addition）：加法的十进制修正。

2. 减法指令

（1）SUB（SUBtract byte or word）：字节或字相减。
（2）SBB（SUBtract Byte or word with carry）：带借位的字节或字相减。
（3）NEG（NEGate byte or word）：字节或字求补。
（4）CMP（CoMPare byte or word）：字节或字的比较。
（5）DEC（DECrement byte or word by l）：字节或字减 l。
（6）AAS（ASCII Adjust for Subtraction）：减法的 ASCII 码修正。
（7）DAS（Decimal Adjust for Subtraction）：减法的十进制校正。

3. 乘法指令

（1）MUL（MULtiple byte or word unsigned）：无符号的字节或字相乘。
（2）IMUL（Integer MULtiple byte or word）：整数字节或字相乘。
（3）AAM（ASCII Adjust for Multiple）：乘法的 ASCII 码修正。

4. 除法指令

（1）DIV（DIVide byte or word unsigned）：无符号的字节或字相除。

（2）IDIV（Integer DIVision signed）：带符号数除法指令。

（3）AAD（ASCII Adjust for Division）：除法的 ASCII 码修正。

（4）CBW（Convert Byte to Word）：字节转换为字。

（5）CWD（Convert Wrord to Double word）：字转换为双字。

4.2.3 位操作类指令

位操作类指令对二进制数的各个位进行操作，包括 3 组指令：逻辑运算指令、移位指令和循环移位指令。

1. 逻辑运算指令

（1）NOT（‘NOT’ byte or word）：字节或字求反。

（2）AND（‘ANDv’ byte or word）：字节或字的逻辑与。

（3）OR（‘inclusive OR’ byte or word）：字节或字的逻辑或。

（4）XOR（‘eXclusive OR’ byte or word）：字节或字的逻辑异或。

（5）TEST（TEST byte or word）：字节或字的测试。

2. 移位指令

（1）SHL/SAL（SHift Logical/Arithmetic Left byte or word）：字节或字的逻辑/算术左移。

（2）SHR（SHift logical Right byte or word）：字节或字的逻辑右移。

（3）SAR（Shift Arithmetic Right byte or word）：字节或字的算术右移。

3. 循环移位指令

（1）ROL（ROtate Left byte or word）：字节或字的循环左移。

（2）ROR（ROtate Right byte or word）：字节或字的循环右移。

（3）RCL（Rotate through Carry Left byte or word）：字节或字通过进位的循环左移。

（4）RCR（Rotate through Carry Right byte or word）：字节或字通过进位的循环右移。

4.2.4 控制转移类指令

控制转移类指令通过修改 CS 和 IP 寄存器的值来改变程序的执行顺序，包括 5 组指令：无条件转移指令、条件转移指令、循环指令、子程序指令和中断指令。

1. 无条件转移指令

JMP（JuMP）：转移。

2. 条件转移指令

条件转移指令共 16 种，详细指令功能如表 4.1 所示。

表 4.1 条件转移指令

助记符	标志位	说　明	助记符	标志位	说　明
JZ/JE	ZF=1	等于零/相等	JC/JB/JNAE	CF=1	进位/低于/不高于等于
JNZ/JNE	ZF=0	不等于零/不相等	JNC/JNB/JAE	CF=0	无进位/不低于/高于等于
Js	SF=1	符号为负	JBE/JNA	CF=1 或 ZF=1	低于等于/不高于

续表

助记符	标志位	说　明	助记符	标志位	说　明
JNS	SF=0	符号为正	JNBE/JA	CF=0 且 ZF=0	不低于等于/高于
JP/JPE	PF=1	“1”的个数为偶	JL/JNGE	SF≠OF	小于/不大于等于
JNP/JPO	PF=0	“1”的个数为奇	JNL/JGE	SF=OF	不小于/大于等于
JO	OF=1	溢出	JLE/JNG	ZF≠OF 或 ZF=1	小于等于/不大于
JNO	OF=0	无溢出	JNLE/JG	SF=OF 且 ZF=0	不小于等于/大于

其中“高于和低于”适用于操作数为无符号数的判别。例如，OFFH 高于 05H。而“大于和小于”适用于操作数为带符号数的判别。例如，OFFH 小于 05H，因为前者为-1。条件转移指令常在比较或减法指令之后使用。

3. 循环指令

（1）LOOP：循环标号。

（2）LOOPE/LOOPZ（LOOP if Equal/Zero）：若相等/为 0，则循环。

（3）LOOPNE/COOPNZ（LOOP if Not Equal/Not Zero）：若不相等/不为 0，则循环。

（4）JCXZ（Jump if register CX=O）：若寄存器（CX）=0，则转移。

4. 子程序指令

（1）CALL（CALL a procedure）：调用一个过程指令（转子指令）。

（2）RET（RETurn from procedure）：从过程返回（从子程序返回）。

5. 中断指令

（1）INT n：中断调用指令，产生 n 号中断。

（2）IRET：中断返回指令，实现中断返回。

（3）INTO：溢出中断指令，若溢出 OF=1，产生 4 号中断，否则顺序执行。

4.2.5　字符串处理类指令

字符串处理类指令包括如下 5 种基本操作。

（1）MOVS/MOVSB/MOVSW（MOV Byte or Word String）：传送字节串或字串。

（2）CMPS/CMPSB/CMPSW（COMPare Byte or Word String）：比较字节或字串。

（3）SCAS/SCASB/SCASW（SCAn Byte or Word String）：搜索字节串或字串。

（4）STOS/STOSB/STOSW（STOre Byte OR Word String）：存字节串或字串。

（5）LODS/LODSB/LODSW（LOaD Byte or Word String）：取字节串或字串。

以上各条指令前可使用相应的重复前缀指令，重复前缀指令如下。

REP/REPE/REPZ/REPNZ（REPeat string operation）。

4.2.6　处理器控制类指令

处理器控制类指令主要有 6 个，它们具体信息如下。

1. 空操作指令

NOP：空操作指令。

2. 段超越前缀指令

SEG：即 CS:、SS:、DS:、ES:、取代默认段寄存器。

3. 标志操作

（1）CLC：清进位标志。

（2）CMC：进位位取反。

（3）STC：进位位置 1。

（4）CLD：清 DF 标志。

（5）STD：置 DF 标志为 1。

（6）CLI：清 IF 标志为 0（关中断）。

（7）STI：置 IF 标志为 1（开中断）。

4. 处理器暂停指令

HLT：暂停。

5. 处理器等待指令

WAIT：等待。

6. 处理器交权指令

ESC：交权。

7. 总线封锁前缀 LOCK

LOCK：总线封锁前缀。

4.3 操作数的类型

Intel 8086/8088 CPU 的指令系统中所需要的操作数有如下 4 种。

（1）立即数操作数：操作数在指令中直接给出。

（2）寄存器操作数：操作数在 CPU 的通用寄存器或段寄存器中。

（3）存储器操作数：操作数在存储器的存储单元中。

（4）I/O 操作数：操作数在输入/输出端口的寄存器中。

单操作数指令的操作数只能是寄存器操作数或存储器操作数，双操作数指令的目标操作数只能是寄存器操作数或存储器操作数，而源操作数可以是四者之一，但是两操作数不能同时为存储器操作数。

在本节中先介绍前三种操作数，I/O 操作数和 I/O 端口寻址在后面章节中介绍。

4.3.1 立即数操作数

立即数操作数（简称立即数）作为机器指令的一部分出现在双操作数指令中。除了乘法、除法和字符串操作指令之外，立即数操作数均可作为源操作数。立即数操作数在汇编指令中可以以十六进制、八进制、二进制和十进制形式书写。

例如：

```
0B2H                ；十六进制形式书写
435Q                ；十进制形式书写
```

```
110B                ；二进制形式书写
74D                 ；十进制形式书写
```

注意：

（1）在用十六进制书写时，如果第一个字符是非数字 0～9，前面一定要补一个 0，例如，CAH 应记为 0CAH。

（2）汇编指令中立即数操作数还可以以一个表达式的形式出现，此时该立即数就是表达式的值。例如，汇编指令“MOV DL,41H”的机器指令是 B241H，其中 B2 是操作码，41 是字节立即数。可见立即数是作为机器指令的一个字段出现在机器指令中的。

（3）立即数只能作为源操作数，不能放在目的操作数位置。

【例 4.2】　编写立即数操作数指令的汇编语言。

代码如下：

```
MOV AX, 1234H       ；1234H 为立即数
```

说明：

汇编指令“MOV AX,1234H”的机器指令是 B83412H，这是一个 3 字节的指令，B8 是操作码，3412H 是字立即数。执行该指令时，立即数 1234H 作为源操作数送入寄存器 AX 中。

4.3.2　寄存器操作数

寄存器操作数是指将操作数存放在寄存器中，寄存器用来存放参与运算的数据或运算的结果。上面代码“MOV AX,1234H”中的 AX 就是寄存器操作数。其中寄存器包括“段寄存器”、“控制寄存器”和“通用寄存器”。

1. 段寄存器

段寄存器是存放段地址的，要将段地址送入段寄存器，必须用通用寄存器作为中间桥梁，即通过通用寄存器将数据传送到 DS、ES 或 SS 段寄存器中。

如果需要将立即数置入段寄存器中的 DS、ES 或 SS，则首先应将该值送入 AX 或其它的通用寄存器，然后再由寄存器传送给 DS、ES 或 SS。至于 CS 因其与指令地址有关，一般不需用户干预。

2. 控制寄存器

控制寄存器包括“指令指针寄存器 IP”和“标志寄存器 FLAGS”，控制寄存器也不能直接传送操作数。

标志寄存器 FLAGS 的标志位在执行算术逻辑运算指令后一般会被修改，至于修改成什么值将依据指令及执行的结果而定，它反映了当时处理器所检测到的结果。标志寄存器一般不能作为操作数。

3. 通用寄存器

通用寄存器（AX、BX、CX、DX、SP、BP、SI、DI、AH、AL、BH、BL、CH、CL、DH、DL）和指针及变址寄存器（BX、BP、SI、DI）均可参加算术和逻辑运算操作。虽然经常把 AX 作为累加器，但所有通用寄存器均可用做累加器。通用寄存器在单操作数指令中可作为目标操作数，在双操作数指令中既可作为源操作数也可作为目标操作数。

【例 4.3】 将段地址 2000H 送入段寄存器中。

代码如下：

```
MOV     AX, 2000H       ；立即数 2000H 送入通用寄存器 AX 中
MOV     DS,AX           ；将段地址 2000H 送入段寄存器 DS 中
```

注意：

在双操作数指令中，允许目标操作数和源操作数都是寄存器，但类型必须一致，即要么都是字节寄存器，要么都是字寄存器。

4.3.3 存储器操作数

存储器操作数又称为内存操作数，是把内存某地址存放的字节、字作为指令的处理对象。这时要将该字节、字作为源操作数或目标操作数，当作为源操作数时从内存中取出，或送到某个寄存器，或参加运算等；而当作为目标操作数时，则是将操作的结果置入该内存单元。我们知道，8086 的存储器是分段管理的，所有这里给出的地址只是偏移地址（称有效地址 EA），而段地址在默认的或用段超越前缀指定的段寄存器中。

无论是哪种操作数，关键是必须找到其所在地址，也就是说必须指出其所在段和相对于段首的位移才能确定其物理地址。

存储器操作数所在段的段寄存器的名字在汇编指令中一般是不写出的，根据存储器操作数类型的不同相应使用不同的段寄存器。

【例 4.4】 存储器操作数。

代码如下：

```
MOV   SI, [2000H]
MOV   [BX+SI+02],AX
```

说明：

这些操作数实际被存放于内存储器中，被操作或运算时需要从存储器中取出并被 CPU 接受送到运算器参与操作运算。

4.4 立即寻址方式

在立即寻址方式下，指令所需要的操作数紧跟在操作码之后。指令在汇编成机器代码后，操作数作为指令的一部分存放在操作码之后的存储单元。它是紧跟在指令操作码后面的一个可用 8 位或 16 位二进制补码表示的有符号数，即操作数的存放地址就是指令操作码的下一个单元，这种操作数被称为立即数，立即数寻址方式只能用于源操作数。

立即数主要用于表示常数，立即数寻址方式常用于给寄存器或存储单元赋值，也可以与寄存器或存储单元中的内容进行算术逻辑运算。

【例 4.5】 将立即数 24H 传给 AL 寄存器中。

代码如下：

```
MOV     AL, 24H         ；AL←24H，机器码为 B024
```

说明：

这段代码是将立即数 24H 传给 AL 寄存器，汇编成机器代码为“B024”，其中 B0 为该指令的操作码，紧跟其后的 24 就是立即数 24H。

【例 4.6】　将立即数 1234H 传给 AX 寄存器中。

代码如下：

```
MOV     AX, 1234H       ；AX←1234H，机器码为 B83412
```

说明：

（1）这段代码是将立即数 1234H 传给 AX 寄存器中，汇编成机器代码为“B83412”，其中 B8 为该指令的操作码，紧跟其后的 3412 就是立即数 1234H。

（2）立即数可以是 8 位的，也可以是 16 位的。如果是 16 位立即数，那么它是按“高高低低”原则存放的，即高位字节存放在高地址存储单元，低位字节存放在低地址存储单元，如图 4.1 所示。

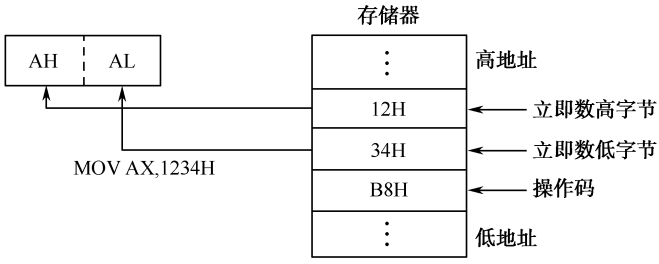


图 4.1　立即数寻址方式

由于在指令执行过程中，立即数作为指令的一部分直接从总线接口单元 BIU 的指令队列中取出，不需另外占用总线周期，因此立即数寻址方式执行速度较快。采用立即数寻址方式不仅能简化数据的存取，而且可使指令的书写直观清晰，因此在程序设计中立即数寻址方式使用得相当多。

4.5　寄存器寻址方式

在寄存器寻址方式下，指令所需的操作数来自 CPU 的某个或某两个通用寄存器，它可以是 8 位寄存器，也可以是 16 位寄存器。8 位寄存器操作数可以是 AL、AH、BL、BH、CL、CH、DL、DH，16 位的寄存器操作数可以是 AX、BX、CX、DX、SI、DI、SP 和 BP。另外操作数还可以存放在 4 个段寄存器 CS、DS、SS 和 ES 中。

由于存取操作数就在 CPU 内部进行，不需要访问存储器而执行总线周期，所以执行速度非常快。另外，通用寄存器数目较少，编码所需的二进制位也就少，因而使用寄存器寻址可以有效地减少指令宽度。因此对于那些需要经常存取的操作数，在程序设计中尽可能采用寄存器寻址方式。

【例 4.7】　寄存器寻址方式。

代码如下：

```
MOV     AX, 1234H       ；AX←1234H
```

```
MOV     BX, AX          ; BX←AX
```

说明：

（1）第一条指令源中源操作数为立即数寻址，目的操作数为寄存器寻址。

（2）第二条指令中两个操作数均为寄存器寻址。

（3）执行上面第一条指令后，AX 为 1234H。第二条指令后，BX 中内容保存不变，也是 1234H，指令执行情况如图 4.2 所示。

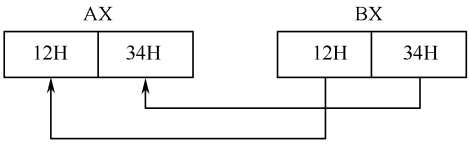


图 4.2　指令“MOV AX,BX”的执行情况

寄存器寻址方式的表示形式是，在汇编指令中直接书写寄存器名，这些寄存器可以是 8 位的，也可以是 16 位的。

注意：

在段寄存器中，CS 寄存器比较特殊，它只能作为源操作数，而不能作为目的操作数，在汇编语言中，CS 内容的改变是通过指令的执行来设置的。

例如：

```
MOV     CS, AX          ; 错误!
MOV     AX, CS          ; 正确!
```

在寄存器寻址方式中，寄存器可以是 8 位的，也可以是 16 位的，在双操作数指令中，寄存器的位数需要跟另一个操作数的位数一致。

例如：

```
MOV     AH, BX          ; 错误!
ADD     CL, BX          ; 错误!
```

4.6　存储器寻址方式

寄存器寻址和立即寻址方式虽然速度很快，但是 CPU 中的寄存器毕竟是有限的，在程序设计中不可能把所有需要的数据都存放在寄存器或者直接存放在指令中。在多数情况下，程序所需要的数据存放在内存中。如何获得内存中的数据和如何把运算结果存储到内存中，称为存储器寻址方式，也称为内存寻址方式。

在存储器寻址方式下，指令的操作数一般都是代码段以外的数据段、堆栈段和附加段中的存储单元，指令给出的是存储单元地址或产生存储单元地址的表达式。在汇编语言源程序中，一个存储单元的地址都是用逻辑地址来表示的，即段基值：偏移量。其中段基值在某个段寄存器中，偏移量表示了该存储单元与段基址之间的距离（字节数），它是由 CPU 的执行单元 EU 来确定和计算的有效地址 EA。执行指令时，CPU 首先根据操作数字段提供的地址信息，由执行单元 EU 计算出有效地址 EA，再由总线接口单元 BIU 根据如下公式计算出操作数的物理地址。

公式：

物理地址=段基值×10H+EA

其中，有效地址 EA 最多可以由 3 个地址分量组合而成。

（1）位移量（Displacement）：是指令中的一个 8 位或 16 位二进制数，通常在源程序中以操作数名字（变量名或标号）的形式出现。汇编以后，这些操作数名字的偏移量转换为指令代码中的位移量。程序设计人员可以修改它的值（某变量名加/减一个常数），也可以用一个常数直接指定一个位移量。

（2）基址（Base Address）：由基址寄存器 BX 或基址指针 BP 提供的内容。

（3）变址（Index Address）：由源变址寄存器 SI 或目的变址寄存器 DI 提供的内容。

这 3 个地址分量组合时，如果有两个或两个以上分量时，则需进行以 2 为模的 16 位二进制数加法运算。正是因为这 3 个地址分量的不同组合，才形成了几种不同的存储器寻址方式。

由于程序所需要的数据大部分存放在内存中，因此存储器寻址非常重要。一般地说，双操作数指令的源操作数和目的操作数不能同时为存储器操作数。存储器寻址方式按 EA 计算方法的不同可分为直接寻址方式、寄存器间接寻址方式、寄存器相对寻址方式、基址变址寻址方式、相对基址变址寻址方式和字符串操作寻址 6 种。

【例 4.8】　存储器寻址方式。

代码如下：

```
MOV     AL,DS:[5]
```

说明：

代码中 DS:[5]就是直接寻址方式，该指令的功能是将数据段位移为 5 的那个内存单元存放的一个字节送入 AL 寄存器中。

4.6.1　直接寻址方式

直接寻址方式是内存寻址方式中最简单的一种，在直接寻址方式下，指令中所需要的操作数是内存操作数，指令中直接给出了该操作数在存储单元中的有效地址 EA，EA 只由位移量组成，即 EA=位移量，位移量是直接包含在指令中的，和指令操作码一起存放在代码段中，而操作数则存放在数据段中。直接寻址方式仅涉及到段寄存器，主要用于存取简单变量。直接寻址方式如图 4.3 所示。

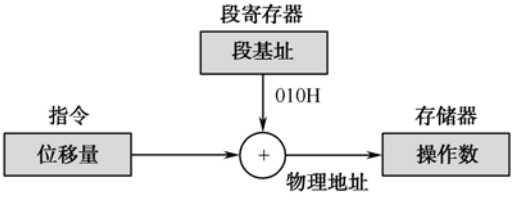


图 4.3　直接寻址方式

在汇编语言中，有效地址 EA 通常用“[ ]”括起来表示，否则指令将变成立即数寻址，而不是直接寻址。

例如：

```
MOV   AX,[3000H]
```

在此例中，指令的功能是将内存单元操作数传送给 AX 寄存器。该指令给出了有效地址 3000H，但它不是存储单元的物理地址。在默认的情况下，有效地址要与数据段寄存器 DS 一起构成操作数所在存储单元的物理地址。在汇编语言程序设计中用 EA 表示存储单元的有效地址，则在直接寻址方式中，存储单元的物理地址为“（DS）*10H+EA”。

位移量可以用带方括号的常数表示，当然也可以用变量表示。

例如：

```
MOV AX，DATA
```

在此例中，DATA 是变量，此指令是把由 DATA 所指的字单元内容传送给寄存器 AX。如果要把字节变量 DATA 所指的字节单元内容传送给寄存器 AL，则代码如下：

```
MOV AL，DATA
```

若操作数不在数据段中，而在代码段、堆栈段或附加段中，则应在操作数地址之前使用段超越前缀指出段寄存器名。

例如：

```
MOV AX,ES:[3000H]              ；AX←ES:[3000H]
```

在此例中，指令是把附加段 ES 中偏移 3000H 个字节的存储单元内容传送给寄存器 AX。段超越前缀也可以用于其他存储器寻址方式中，指明给定的段寄存器替代了默认的段寄存器。此时操作数的在内存中的物理地址的为“(ES)*10H+EA”，其中 EA=3000H。

【例 4.9】 直接寻址方式。

代码如下：

```
MOV     BX,DATA
MOV     AX, [200H]
MOV     CX, ES：[300H]
```

说明：

（1）在第一条指令中，DATA 为变量，此指令是把由 DATA 所指的字单元内容传送给寄存器 BX，等价于“MOV BX, DS：DATA”。

（2）在第二条指令中，该指令给出了有效地址 200H，功能是将当前数据段偏移 200H 个字节的字存储单元内容传送给寄存器 AX，等价于“MOV AX, DS：[200H]”。设(DS)=2000H，那么要访问的字存储单元的物理地址为：2000H×10H+200H= 20200H，如果该字存储单元的内容为 6738H，即有(20200H)=6738H，指令执行的结果将是 20200H 字存储单元内容 6738H 传送给寄存器 AX，即(AX)=6738H，而 20200H 的内容保持不变，其执行情况如图 4.4 所示。

（3）在第三条指令中，该指令给出了有效地址 300H，功能是将内存单元((ES)*10H+300H)操作数传送给 CX 寄存器。

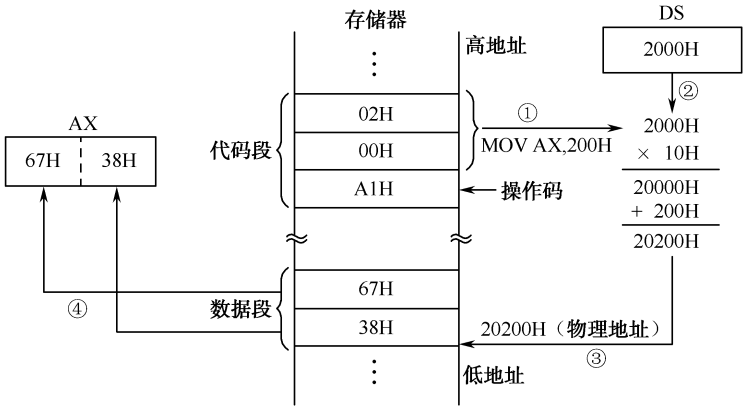


图 4.4　指令“MOV AX,[200H]”执行情况

4.6.2　寄存器间接寻址方式

寄存器间接寻址方式与寄存器直接寻址方式不同，寄存器间接寻址方式不是把寄存器的内容作为操作数，而是作为有效地址 EA。此时寄存器可以是 SI、DI、BP、BX 之一，在书写汇编指令时相应寄存器名用[]括起来，表示该寄存器的内容为 EA。

例如：

```
MOV AX, [BP]                ; AX←((SS)*10H+(BP))
```

此例表示 SS 内容乘以 10H 再加上 BP 的内容构成了物理地址，取该地址的内容送入 AX 寄存器。

在寄存器间接寻址方式中，指令操作数存放在内存单元中，而寄存器存放操作数在内存中的偏移地址。寄存器间接寻址方式是一种把寄存器的内容作为操作数在内存中的有效地址的内存寻址方式，如图 4.5 所示。

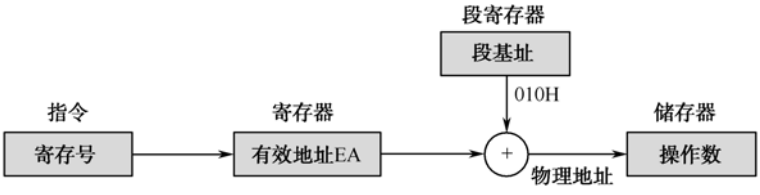


图 4.5　寄存器间接寻址示意图

在寄存器间接寻址方式中，操作数的有效地址由寄存器提供，而操作数的段地址由相应寄存器默认的段寄存器提供。其中，BX、SI、DI 默认的段寄存器为 DS，而 BP 默认的段寄存器为 SS。此物理地址形成规则如表 4.2 所示。

在有些特殊指令中，存储操作数有效地址的寄存器默认的段寄存器可能会发生变化，例如在串操作指令中，通常用 SI 存储源串的有效地址，DI 存储目的操作数的有效地址，SI 和 DI 所对应的段寄存器分别为 DS 和 ES。

表 4.2 寄存器间接寻址物理地址的形成

变址寄存器	汇编书写形式	有效地址（EA）	段寄存器	物理地址
SI	[SI]	(SI)	DS	(DS)*10H+(SI)
DI	[DI]	(DI)	DS	(DS)*10H+(DI)
BX	[BX]	(BX)	DS	(DS)*10H+(BX)
BP	[BP]	(BP)	SS	(SS)*10H+(BP)

在寄存器间接寻址方式中，也可以通过段前缀指定寄存器对应的段寄存器。

例如：

```
MOV     AX , ES:[BX]        ; AX←((ES)*10H+(BX))
MOV     ES:[SI],1234H       ; ((ES)*10H+(SI)) ←1234H
```

如果指令中指定了段超越前缀，那么计算操作数的物理地址时，应由指定的段寄存器提供段基值。使用寄存器间接寻址的表示形式是指令中寄存器名加方括号。

例如：

```
MOV     AX,[BX]             ; 等价于 MOV AX,DS:[BX]
MOV     DL,[BP]             ; 等价于 MOV DL,SS: [BP]
MOV     [DI],CX             ; 等价于 MOV DS:[DI],CX
MOV     BH,[SI]             ; 等价于 MOV BH,DS:[SI]
MOV     DS：[BX],AL         ; 等价于 MOV [BX],AL
```

【例 4.10】 假设(DS)=1000H，(DI)=7834H，存储单元 17834H 的内容是 4832H，执行 MOV BX,[DI]指令后，BX 寄存器的内容是什么？

根据寄存器间接寻址方式的规则，在执行本例指令时，寄存器 DI 的值不是操作数，而是操作数的地址。该操作数的物理地址应由 DS 和 DI 的值组成，即物理地址 PA 为：

PA=(DS)*10H+DI=1000H*10H+7834H=17834H。

所以，该指令的执行时是把从物理地址为 17834H 开始的一个字的值传送给 BX，所以执行后(BX)=4832H，其执行情况如图 4.6 所示。

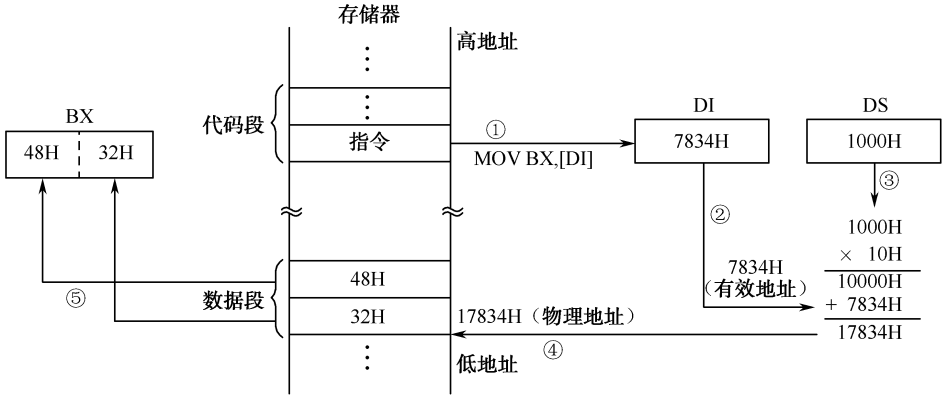


图 4.6 指令“MOV BX,[DI]”的执行情况

由图可见，寄存器 DI 给出要访问的存储单元的有效地址 EA。如果操作数存放于其他段，如附加段，那么指令将改写为：MOV BX, ES:[DI]。

说明：

从例中可以看出，寄存器间接寻址方式实际上是将有效地址 EA 事先存放在一个寄存器中。如果在程序运行期间对间接寻址寄存器的内容进行修改，那么就实现了同一条指令对不同存储单元的操作和处理。

这个间接寻址寄存器就像一个地址指针，可以将数组的起始地址存放在间接寻址寄存器中，因为数组的元素长度相同，所以只需在执行完一条指令后改变间接寻址寄存器的内容（增加或减少间接寻址寄存器中的地址），就可以用同一个地址表达式来指定一维数组中的不同元素，对相邻数组元素进行操作，从而实现对连续存储空间中的不同存储单元进行操作和处理，因此寄存器间接寻址方式常常用来处理一维数组。另外，寄存器间接寻址方式也常常应用于循环程序设计中。

注意：

在书写寄存器间接寻址的指令时，间接寻址寄存器名一定要放在方括号中，否则寻址方式可能完全不同。

例如：

```
MOV     AX，[BX]      ；源操作数为寄存器间接寻址
MOV     AX，BX        ；源操作数为寄存器寻址
```

4.6.3　寄存器相对寻址方式

寄存器相对寻址方式是其操作数在存储器中，有效地址是一个基址寄存器（BX、BP）或变址寄存器（SI、DI）的内容与有符号的 8 位或 16 位位移量之和，如图 4.7 所示。其有效地址的计算公式如下，此物理地址形成规则如表 4.3 所示。

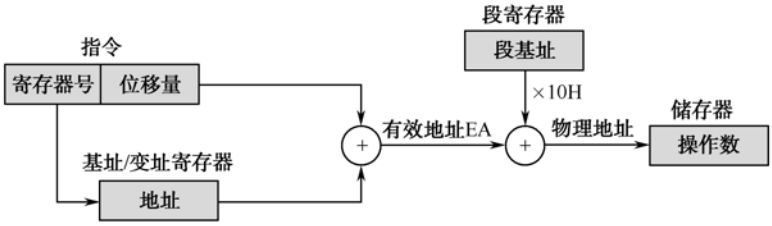


图 4.7　寄存器相对寻址方式示意图

表 4.3　寄存器相对寻址物理地址的形成规则

变址器	汇编书写形式	有效地址（EA）	段寄存器	物理地址
SI	[SI+disp]	(SI) +disp	DS	(DS)*10H+(SI) +disp
DI	[DI+disp]	(DI) +disp	DS	(DS)*10H+(DI) +disp
BX	[BX+disp]	(BX) +disp	DS	(DS)*10H+(BX) +disp
BP	[BP+disp]	(BP) +disp	SS	(SS)*10H+(BP) +disp

注：表中 disp 为位移量。

公式：

有效地址 EA=BX/BP/SI/DI+8/16 位位移量

其中，

（1）计算物理地址时，BX、SI、DI 寄存器默认数据段 DS，BP 寄存器默认堆栈段 SS。若操作数前指定了段寄存器，那么以指定的段寄存器为准。

（2）指令代码中的位移量采用补码表示，如果是 8 位，则被带符号扩展为 16 位。当得到的有效地址超过 FFFFH 时，则取 64KB（即 64×1024）的模。

当使用非默认段时，可用段超越前缀。

例如：

```
MOV AX,[DI+45H]          ；等价于：MOV AX,DS:[DI+45H]
```

这条指令中使用的是 DI 寄存器，位移量为 45H，那么操作数的有效地址 EA=DI+45H，与 DI 寄存器约定的段为数据段 DS。如果是 BP，则默认为堆栈段 SS。

例如：

```
MOV AX,[BP+45H]          ；等价于：MOV AX,SS:[BP+45H]
```

该指令使用的是 BP 寄存器，与之约定的段为堆栈段 SS。另外，采用 BP 相对寻址时，如果偏移量为 0，也可以不写出来，形式上与寄存器间接寻址一样。

例如：

```
MOV AX，[BP]             ；等价于：MOV AX，[BP+0H]
```

书写时基址寄存器名或变址寄存器名一定要放在方括号中，而位移量可以不写在方括号中。

例如：

```
MOV AX, 20H[SI]          ；等价于：MOV AX, [SI+20H]，MOV AX,DS:[SI+20H]
```

在寄存器相对寻址指令中，可以含有变量，作为位移量。

例如：

```
MOV DX, DATA[BP]         ；等价于：MOV DX, [BP+DATA]，MOV DX,SS:[SI+DATA]
MOV DATA[DI],CX          ；等价于：MOV [DI+DATA],CX，MOV DS: DATA [DI],CX
```

以上两条指令中，DATA 为变量。

【例 4.11】 假设(DS)=1000H，(DI)=7834H，存储单元 17854H 的内容是 4832H，执行 MOV BX,20H[DI]指令后，BX 寄存器的内容是什么？

根据寄存器相对寻址方式的规则，在执行本指令时，寄存器 DI 的值不是操作数，而是操作数的地址，所以源操作数的有效地址 EA 为：

EA=(DI)+20H=7834H+20H=7854H

该操作数的物理地址应由 DS 和 EA 的值形成，即物理地址 PA 为：

PA=(DS)*10H+EA=1000H*10H+7854H=17854H。

所以，该指令的执行效果是把从物理地址为 17854H 开始的一个字的值传送给 BX，所以(BX)=4832H，其执行情况如图 4.8 所示。

由图可见，寄存器 DI 的值不是操作数，而是操作数的地址，再和指令中偏移量 20H 相加得到有效地址 EA。

说明：

从例子指令执行情况图中可以看出，寄存器相对寻址方式不仅可以像寄存器间接寻址那样通过修改寄存器的内容来访问不同的存储单元，而且这种寻址方式中还增加了一个“位移量”分量，可在处理数组和表格时更加方便，用位移量指定数组或表格的首地址，

用一个基址寄存器或变址寄存器指向数组或表格中的某一个元素，那么寄存器中的内容就是数组或表格首地址距离要访问的数组或表格元素的字节数（从 0 开始计算），即寄存器（BX、BP、SI、DI）中的地址是相对于数组或表格首地址的，也因此称为寄存器相对寻址。寄存器相对寻址方式在程序设计中应用非常广泛。

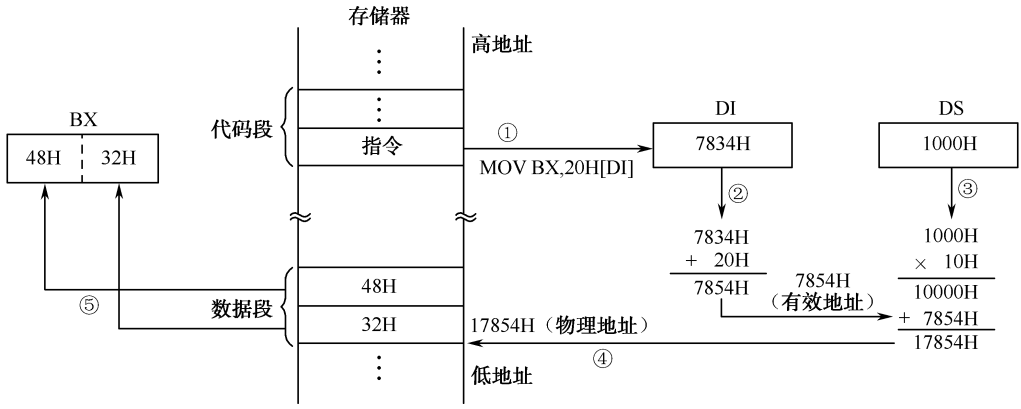


图 4.8　指令“MOV BX,20H[DI]”的执行情况

4.6.4　基址变址寻址方式

基址变址寻址方式同样是通过寄存器来提供指令操作数在内存中的有效地址，但是寄存器基址变址寻址方式和寄存器间接寻址方式不同，是把一个基址寄存器（BX 或 BP）的内容加上变址寄存器（SI 或 DI）的内容构成有效地址 EA。如图 4.9 所示。其有效地址 EA 的计算公式如下，此物理地址形成规则如表 4.4 所示。

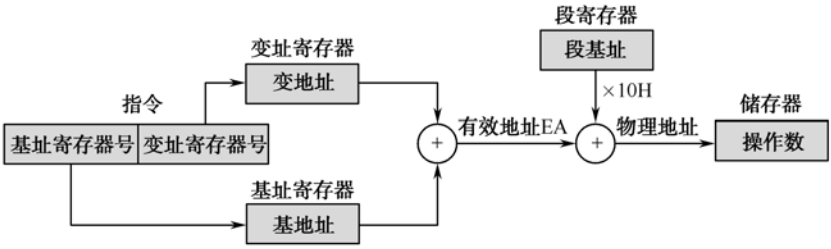


图 4.9　基址变址寻址方式示意图

公式：

有效地址 EA=BX/BP+SI/DI

其中：若基址寄存器使用 BX，其默认段为数据段 DS。若基址寄存器使用 BP，其默认段为堆栈段 SS。在基址变址寻址方式中，基址寄存器和变址寄存器需要用“[ ]”括起来。

例如：

```
MOV   AX,[BX+SI]          ; AX←DS:[BX+SI]
MOV   AX,[BP+DI]          ; AX←SS:[BP+DI]
```

```
MOV   AX,[BX+DI]          ; AX←DS: [BX+DI]
MOV   AX,[BP+SI]          ; AX←SS:[BP+SI]
```

当得到的有效地址超过 FFFFH 时，则取其 64KB 的模。

表 4.4　基址变址寻址物理地址的形成规则

基址寄存器	变址寄存器	有效地址（EA）	段寄存器	物理地址
BX	SI	(BX)+(SI)	DS	(DS)*10H+(BX) +(SI)
BX	DI	(BX)+(DI)	DS	(DS)*10H+(BX) +(DI)
BP	SI	(BP)+(SI)	SS	(SS)*10H+(BP) +(SI)
BP	DI	(BP)+(DI)	SS	(SS)*10H+(BP) +(DI)

基址变址寻址方式允许使用段前缀，若指定了段寄存器，那么以指定的段寄存器为准。例如：

```
MOV   AX,ES:[BX+SI]       ; AX←ES:[BX+SI]
MOV   AX,DS:[BP+SI]       ; AX←DS:[BP+SI]
```

基址变址寻址方式有时可以写成不同的形式。

例如：

```
MOV   AX,[BX][SI]         ; 等价于：MOV AX,[BX+SI]
MOV   AX,[BP+DI]          ; 等价于：MOV AX,[BP]+[DI]
```

【例 4.12】 假设指令为：MOV AX,[BX+SI]，(DS)=1000H，(BX)=4832H，(SI)=7834H，内存单元 1C066H 的内容为 1234H。问该指令执行后，AX 的值是什么？

根据基址变址寻址方式的规则，在执行本例指令时，源操作数的有效地址 EA 为：

EA=(BX)+(SI)=4832H+7834H=C066H

该操作数的物理地址应由 DS 和 EA 的值形成，物理地址 PA 为：

PA=(DS)*10H+EA=1000H*10H+C066H=1C066H

该指令的执行时是把从物理地址为 1C066H 开始的一个字的值传送给 AX，所以执行后，(AX)=1234H，其执行情况如图 4.10 所示。

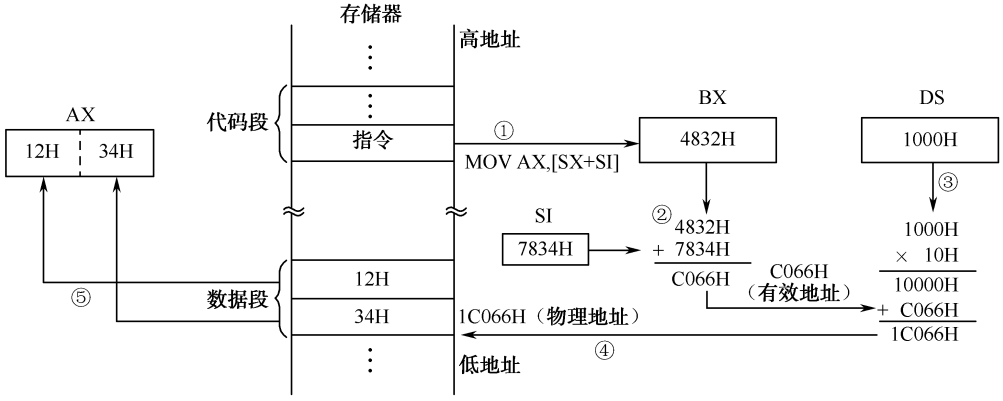


图 4.10　指令“MOV AX,[BX+SI]”的执行情况

说明：

例 4.12 中目标操作数为 AX，如果目标操作数是 BX，即指令为“MOV BX,[BX+SI]”，则执行指令后 BX 的内容是 1234H。

4.6.5　相对基址变址寻址方式

相对基址变址寻址方式和基址变址寻址方式类似，也使用 BX 和 BP 为基址寄存器，以 SI 和 DI 为变址寄存器构成的存储器寻址方式，另外还在指令中指定一个 8 位或 16 位的位移量，这三者之和构成操作数的有效地址 EA，如图 4.11 所示。其有效地址 EA 的计算公式如下，此物理地址形成规则如表 4.5 所示。

公式：

有效地址 EA=BX/BP+SI/DI+8/16 位位移量

其中，

（1）若基址寄存器使用 BX，其默认段为数据段 DS。若基址寄存器使用 BP，其默认段为堆栈段 SS。在相对基址变址寻址方式中，基址寄存器、变址寄存及位移量三者需要用“[ ]”括起来。

例如：

```
MOV AX,[BX+SI+11H]      ；AX←DS:[BX+SI+11H]
```

（2）指令代码中的位移量采用补码表示，如果是 8 位，则被带符号扩展为 16 位。当得到的有效地址超过 FFFFH 时，则取其 64KB 的模。

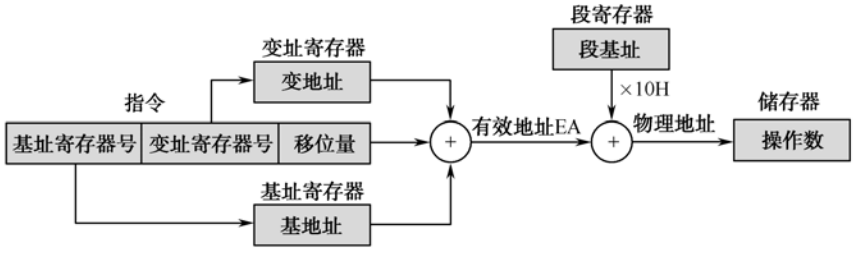


图 4.11　相对基址变址寻址方式示意图

表 4.5　相对基址变址寻址物理地址的形成规则

基址器	变址器	汇编书写形式	有效地址	段寄存器	物理地址
BX	SI	[BX+SI+disp]	(BX)+(SI)+disp	DS	(DS)*10H+(BX) +(SI)+disp
BX	DI	[BX+DI+disp]	(BX)+(DI) +disp	DS	(DS)*10H+(BX) +(DI)+disp
BP	SI	[BP+SI+disp]	(BP)+(SI) +disp	SS	(SS)*10H+(BP) +(SI)+disp
BP	DI	[BP+DI+disp]	(BP)+(DI) +disp	SS	(SS)*10H+(BP) +(DI)+disp

注：表中 disp 为位移量。

相对基址变址寻址方式有时可以写成不同的形式，位移量可以放在方括号前，也可以放在方括号后，还可以同寄存器一起写成一个地址表达式。

例如：

```
MOV   AX,[BX][SI]11H          ；AX←DS:[BX+SI+11H]
```

```
MOV   AX,11H[BX][SI]          ; AX←DS:[BX+SI+11H]
MOV   AX,[BX+SI]11H           ; AX←DS:[BX+SI+11H]
MOV   AX,[BX+SI+11H]          ; AX←DS:[BX+SI+11H]
```

以上四条指令都是等价的。在相对基址变址寻址方式中，位移量可用符号表示。

例如：

```
MOV AX,[BX+SI+DATA]
```

此条指令中 DATA 是事先定义的变量或常量，此处就是一个数值，其中 DATA 可以是 0，这种情况下将省略 DATA，也就成为基址变址寻址方式。前面介绍过存储器寻址中的直接寻址、寄存器间接寻址、基址变址寻址实际上都是相对基址变址寻址的特例。

【例 4.13】 假设指令：MOV AX,[BX+SI+1010H]，(DS)=1000H，(BX)=4832H，(SI)=7834H，内存单元 1D076H 的内容为 1234H。问该指令执行后，AX 的值是什么？

解：根据相对基址变址寻址方式的规则，在执行本指令时，源操作数的有效地址 EA 为：

$$EA=(BX)+(SI)+1010H=4832H+7834H+1010H=D076H$$

该操作数的物理地址应由 DS 和 EA 的值形成，物理地址 PA 为：

$$PA=(DS)*10H+EA=1000H*10H+D076H=1D076H$$

该指令的执行时是把从物理地址为 1D076H 开始的一个字的值传送给 AX，所以执行后，(AX)=1234H，其执行情况如图 4.12 所示。

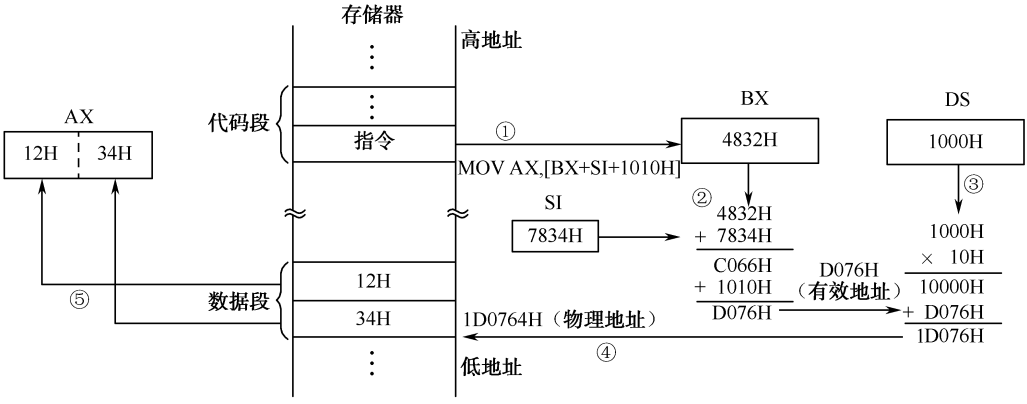


图 4.12 指令“MOV AX,[BX+SI+1010H]”的执行情况

说明：

从相对基址变址寻址这种方式来看，由于它的可变因素较多，有 3 个分量：位移量、基址寄存器、变址寄存器，看起来就显得复杂些，但正因为其可变因素多，它的灵活性也就很高。可以用来处理二维数组或表格，即将二维数组或表格的首地址存放于位移量中，将二维数组或表格元素的行坐标和列坐标做对应处理后，存放在基址寄存器和变址寄存器中，这样就可以通过修改基址寄存器和变址寄存器的内容，用同一个地址表达式来访问数组或表格中的不同元素或表项。

此外，相对基址变址寻址还常常应用于访问堆栈中的数组，一般用 BP 指向栈顶，用位移量表示从栈顶到数组起点的距离（即堆栈内数组的首地址），由变址寄存器指向要访问的数组元素，这样就可以方便地访问和处理堆栈中的数组元素。

4.6.6　字符串操作寻址方式

Intel 8086/8088 CPU 提供了一些专门用于字符串操作的指令，虽然字符串操作指令的操作数在存储器中，它不同于其他指令，它使用隐含的变址寄存器（SI 和 DI）来寻址，且操作过程中自动修改该变址寄存器内容，使其指向下一步操作的地址，修改方式分自动增和自动减两种，采用哪种方式取决于此时状态寄存器的 DF 标志位，当 DF 为 1 时减，当 DF 为 0 时增。其中源操作串由 DS:[SI]指定，目标串由 ES:[DI]指定。

CLD 指令置 DF=0，设置为正向串，每次串操作后 SI 和 DI 自动递增；STD 指令置 DF=1，设置为负向串，每次串操作后 SI 和 DI 自动递减。

当串操作为字节串时，每次 SI 和 DI 递增和递减 1；当串操作为字串时，每次 SI 和 DI 递增和递减 2。

对源操作串有效地址 EA=(SI)，段基址由 DS 指定；对目标串有效地址 EA=(DI)，段基址由 ES 指定，如图 4.13 所示。

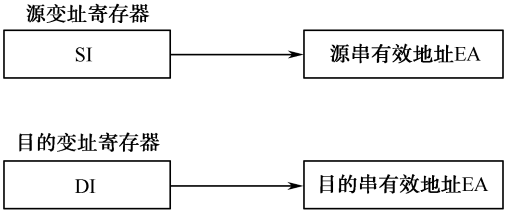


图 4.13　串操作的寻址方式

习题

1．汇编语言指令的格式是怎样的？

2．Intel 8086/8088 CPU 指令系统大致可分为哪几类？

3．什么是寻址方式？8086/8088 有几种寻址方式？各类寻址方式的基本特征是什么？

4．访问内存单元的寻址方式有几种？

5．解释下列术语：操作码、操作数、立即数操作数、寄存器操作数、存储器操作数、段地址、有效地址、物理地址。

6．写出下例指令中，源操作数和目的操作的寻址方式。

（1）MOV　AX,1234H

（2）MOV　AX,BX

（3）MOV　AX,[SI]

（4）MOV　AX,[BX][SI]

（5）MOV　BUFF[BP[SI]],AL

（6）MOV　[SI],AX

（7）MOV　AX,ARRAY[SI]

7．判断以下指令书写形式的正确性。

（1）MOV　AL,CX

（2）MOV　AL,BL

（3）INC　[AX]

（4）MOV　[AX],[SI]

（5）MOV　BL,F6H

8．写出以下汇编指令内存操作数的物理地址的表示方法。

（1）MOV　AL,[BX+5]

（2）MOV　[BP+3],AX

（3）DEC　WORD PRT [SI]

（4）MOV　ARRY[BP][SI],AL

（5）MOV　[1000H],AX

9．假设(DS)=0C08H，(SS)=0C0AH，(ES)=0C0BH，(SI)=1000H，(BX)=0080H，(BP)=100H，变量 VAR 的有效地址为 020H，试指出下列指令中内存操作数的寻址方式，它的有效地址和物理地址分别是多少？

（1）MOV　AX,[1000H]

（2）MOV　BX,ES:[BX]

（3）MOV　VAR[BX],AL

（4）MOV　VAR[BX][SI],1234H

（5）MOV　BX,[BP]

（6）MOV　BX,[BP][SI]

（7）MOV　DS:[BP],AX

第 5 章

汇编语言源程序组成

用汇编语言编制的程序称为汇编语言源程序。汇编语言源程序由汇编语句组成，这些语句不仅包括由汇编指令构成的指令语句，还包括变量定义的说明语句，以及各种伪指令语句。在编写汇编语言源程序时必须遵守汇编语言的规范和约定，具体地说，就是要掌握汇编语言中使用的字符、符号、常数、运算符、表达式及源程序的书写格式等。

本章首先介绍汇编语言源程序的格式，再介绍汇编语言数据、表达式、运算符、常用的伪指令，以及汇编语言源程序返回 DOS 的方法。

5.1 汇编语言源程序的格式

汇编语言源程序由模块、段、子程序和宏等部分组成。一个模块对应一个目标文件，当开发较大型的应用程序时，该程序可能由若干个目标文件或库结合而成。

汇编语言源程序有严格的格式要求，汇编语言源程序格式就是汇编语言必须遵守的语法规则，它有两种书写形式，一种称为简化段定义格式，另一种称为完整段定义格式。

5.1.1 段的定义

微机系统的内存是分段管理的，为了与之相对应，汇编语言源程序也由若干段构成。8086 CPU 主要分为代码段、数据段、堆栈段和附加段四类，在该系统环境下运行的程序在某个时刻最多可访问四个段，而 80386 及其以后的 CPU 都含有六个段寄存器，于是，在这些系统环境下开发的运行程序在某个时刻最多可访问六个段。

不管程序在某个时刻最多能访问多少个段，在编程序时，程序员都可以定义比该段数更多的段。在通常情况下，一个段的长度不能超过 64KB，在 80386 及其以后系统的保护方式下，段基地址是 32 位，段的最大长度可达 4GB。

段的长度是指该段所占的字节数，如果段是数据段，则其长度是其所有变量所占字节数的总和；如果段是代码段，则其长度是其所有指令所占字节数的总和。

在定义段时，每个段都有一个段名。在取段名时，要取一个具有一定含义的段名。书写时，应该首先用段定义伪指令 SEGMENT/ENDS 定义各逻辑段，再用段寻址伪指令

ASSUME 指出各逻辑段与段寄存器之间的关系，各逻辑段的顺序是随意安排的，通常是数据段在前，代码段在后。段定义的一般格式如下。

格式：

```
段名  SEGMENT    [对齐类型]  [组合类型]  [类别]
      …          ;段内的具体内容
      …          ;段内的具体内容
段名  ENDS
```

其中，

（1）段名必须是一个合法的标识符，首尾两个段名要相同。段名可以是唯一的，也可以与其他段同名。在同一模块中，如果有两个段同名，则后者被认为是前段的后续，这样，它们就属同一段。

（2）对齐类型（ALIGN）表示当前段对起始地址的要求，连接程序（LINK.EXE）按表 5.1 的地址格式来定位段的起始地址。在进行段定位时，会根据其定位类型进行定位，所以，各段之间就有可能出现一些空闲字节，也就是说可能浪费几个字节单元。

对齐类型 BYTE 和 WORD 通常用于数据段的定位，对齐类型 DWORD 通常用于 80386 及其以后 CPU 代码段的定位。表 5.1 列出了段对齐类型与段起始地址之间的对应关系。

表 5.1　段对齐类型与段起始地址之间的对应关系

对齐类型	起始地址(二进制)	功能说明	最多空闲字节数
BYTE	xxxx xxxx xxxx xxxx xxxx	下一个字节地址	0
WORD	xxxx xxxx xxxx xxxx xxx0	下一个字地址	1
DWORD	xxxx xxxx xxxx xxxx xx00	下一个双字地址	3
PARA	xxxx xxxx xxxx xxxx 0000	下一个节地址(默认类型)	15
PAGE	xxxx xxxx xxxx 0000 0000	下一个页地址	127

（3）组合类型（COMBINE）是告诉连接程序（LINK.EXE）如何把不同模块中段名相同的段合并在一起。具体的组合类型如表 5.2 所示。

表 5.2　组合类型

组合类型	说　明
NONE	表示当前段在逻辑上独立于其他模块，并有其自己的基地址。NONE 是默认的组合类型
PUBLIC	表示当前段与其他模块中同段名的 PUBLIC 类型段组合成一个段。组合的先后次序取决于 LINK 程序中目标模块排列的次序。在组合时，后续段的起始地址要按其对齐类型进行定位，所以，同名段之间可能有间隔
COMMON	表示当前段与其他模块中同名段重叠，也就是说，它们的起始地址相同。最终段的长度是同名段的最大长度。由于段覆盖，所以，前面同名段中的初始化数据被后续段的初始数据覆盖掉
STACK	组合类型 STACK 表示当前段是堆栈段，其组合情况与 PUBLIC 相同
AT 数值表达式	该数值表达式是当前段所指定的绝对起始地址的段地址
MEMORY	表示本段在存储器中应定位在所有其他段的最高地址上。如果有多个组合类型为 MEMORY 的逻辑段，则只将第一个遇到的段当做 MEMORY 处理，其他段都按 COMMON 处理

（4）（类别（CLASS）是一个由程序员指定的用单引号括起来的字符串，但是它不能与程序中的变量、标号或其他标识符或关键字相同，在段定义伪指令中必须用单引号括起来。如果一个段没有给出类别，那么，这个段的类别就为空。类别主要用于段的分类，在程序连接时，不同模块、不同名的逻辑段只要类别名相同，就放在一个连续的物理存储空间，如果组合类型不是 PUBLIC 或 COMMON，那么每个逻辑段之间是独立的，不进行组合。大多数汇编程序使用“CODE”、“DATA”和“STACKL”等来分别作为代码段、数据段和堆栈段的类别名，以便使源程序的代码和数据等连续。

用段定义伪指令 SEGMENT/ENDS 定义一个逻辑段时，必须为这个段设置段名，而定位类型、组合类型和类别 3 个参数是任选的，各参数之间用空格分隔，但要注意的是，在任选参数时，可以只选其中一个或两个参数，但不能交换它们之间的顺序。

下面是一个分段结构的源程序框架：

```
STACKl      SEGMENT    PARA    STACK    'STACK0'
            …
STACKl      ENDS
DATAl       SEGMENT    PARA    'DATA'
…
DATAl       ENDS
STACK2      SEGMENT    PARA    'STACK0
…
STACK2      ENDS
DATA2       SEGMENT    PARA    'DATA'
…
DATA2       ENDS
CODE        SEGMENT    PARA    MEMORY
            …
START:      …
CODE        ENDS
            END        START
```

上例中源程序汇编和连接以后，程序装入内存的情况如图 5.1 所示。由于各逻辑段的定位类型都选用 PARA，所以段与段之间有可能存在“空白”（用灰色表示）。代码段的组合类型是 MEMORY，所以它位于所有逻辑段的最高存储地址。对于不太大的程序，例如初学者的练习程序，通常只需要 3 个段分别来作为代码段、数据段和堆栈段就可以了。

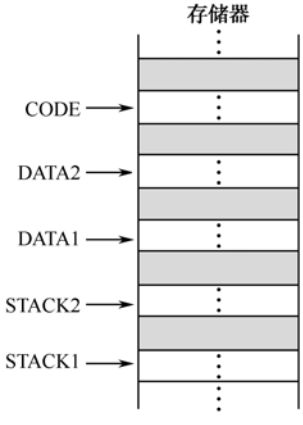


图 5.1　程序装入内存情况

例如：

```
DATA        SEGMENT
            …
DATA        ENDS
STACK       SEGMENT      STACK
            …
STACK       ENDS
CODE        SEGMENT
```

```
            …
START:      …
CODE        ENDS
            END         START
```

注意：

当同一模块中出现两个同名段时，则后者的可选项属性要么与前者相同，要么不写其属性而选用前者的段属性。

通常情况下，代码和数据分别存放，但它们也可以合并在同一个段中。一个完整的汇编语言源程序至少应该含有一个代码段，但一个汇编语言源程序模块却可以只有数据段。另外，一个完整的汇编语言源程序应设置有自己的堆栈段，如果没有，操作系统 DOS 在装载程序时会指定一个堆栈段。由于堆栈段的安排比较固定，为了简单，在程序较小时可以直接利用 DOS 安排的堆栈段。尽管连接程序会在连接时对没有堆栈的目标文件发出一条警告信息，但可以忽略它。对于不太复杂的程序，通常只要 3 个段就可以了。

【例 5.1】 在屏幕上输出“Hello World!”的汇编语言源程序。

程序如下：

```
DATA    SEGMENT                     ; 定义数据段,第一个数据段
NOTE    DB      'Hello '
DATA    ENDS                        ; 第一个数据段结束
STACKl  SEGMENT STACK               ; 定义堆栈段
        DB      40H  DUP(0)         ; 设置堆栈大小为 40H 字节
STACKl  ENDS
CODE    SEGMENT                     ; 定义代码段,第一个代码段
        ASSUME  CS:CODE, DS:DATA    ; 将代码段指定给 CS 寄存器，将数据段指定给 DS 寄存器
BEGIN:  MOV     AX,DATA             ; AX 累加器
        MOV     DS,AX               ; DS 数据段寄存器
        MOV     DX,OFFSET NOTE      ; 这句可以用 LEA DX,NOTE 来替代，DX 是数据通用寄存器
        MOV     AH,09H              ; 09H 对应的是显示字符串
        INT     21H                 ; INT 21H 是用来调用功能的，具体区别用 AH 来识别
CODE    ENDS                        ; 第一个代码段结束
DATA    SEGMENT                     ; 定义数据段,第二个数据段
        DB      'World!',0ah,0dh,'$'
DATA    ENDS                        ; 第二个数据段结束
CODE    SEGMENT                     ; 定义代码段,第二个代码段
        MOV     AH,4CH              ; 4CH 功能在于让你退出程序，
        INT     21H                 ; 调用功能，由上面的 AH 决定，这句就是要退出程序了
CODE    ENDS                        ; 第二个代码段结束
        END     BEGIN
```

说明：

（1）在例 5.1 中，第二个数据段是第一个数据段的后续，汇编程序把它们合二为一。代码段也如此，第二个代码段是第一个代码段的后续，汇编程序把它们合二为一。

（2）例 5.1 中的字段属性使用默认值。

5.1.2　简化段定义格式

简化段定义书写源程序时格式简单，易于掌握。在简化段定义的格式中，以圆点开始的伪指令说明程序的结构。简化段定义格式如下。

格式：

```
.MODEL   SMALL                ; 定义程序的存储模式（一般采用 SMALL）
.STACK   nH                   ; 定义堆栈段，其中 n 为以字节为单位的栈长
.DATA                         ; 定义数据段
…                             ; 定义数据
.CODE                         ; 定义代码段
START:   MOV AX,@DATA
         MOV DS,AX            ; 装填数据段寄存器
         …                    ; 程序代码
         MOV AX,4COOH
         INT 21H              ; 程序结束点，返回 DOS
         …                    ; 子程序代码
         END START            ; 汇编结束，程序起始点为标号 START 处
```

其中，

（1）**.**DATA、**.**CODE 和**.**STACK 依次说明数据段、代码段和堆栈段，一个段开始自动结束上一个段定义。

（2）**.**MODEL 伪指令用来指定该汇编程序所使用的内存模式，内存模式说明了程序中的数据及代码的默认长度，且使用简化段定义的程序必须用 MODEL 指出内存模式。各种内存模式的含义如表 5.3 所示。

表 5.3　内存模式

存储模型	功　能	适用操作系统
TINY（最小模式）	所有数据和代码都放在一个小于 64KB 的段内，访问都为 NEAR 型，并会产生.COM 文件	MS-DOS
SMALL（小模式）	代码在一个 64KB 段中，数据在另一个 64KB 段中，通常编写的汇编程序使用最小模式就已经足够	MS-DOS Windows
MEDIUM（中模式）	代码可以大于 64KB，转移或调用可为 FAR 型，但数据只能在一个单一的 64KB 段中	MS-DOS Windows
COMPACTt（紧凑模式）	代码在一个单一的 64KB 段中，转移或调用可为 NEAR 型，而数据可以大于 64KB	MS-DOS Windows
LARGE（大模式）	代码和数据都可以大于 64KB，但任何数组不能大于 64KB，数据和代码的访问类型都是 FAR 型	MS-DOS Windows
HUGE（特大模式）	代码和数据都可以大于 64KB，数组也可以超过 64KB	MS-DOS Windows
FLAT (平展模式)	所有代码和数据放置在一个段中，但段地址是 32 位的，所以整个程序可为 4GB	OS/2 WindowsNT

（3）.STACK 定义堆栈段的开始，其中“nH”可以不写，其默认值为 1024B，为程序开辟 1024 字节的堆栈空间。

（4）.DATA 说明其下面的内容是数据段中的变量定义。一个源程序可以有多个伪指令.DATA 定义的数据段，这就好像在源程序中定义多个同段名的数据段一样。另外，还有两个很少使用的伪指令来定义数据段.DATA?/.CONST，伪指令.DATA?说明下面是一个未初始化数据段的开始；伪指令.CONST 说明下面是一个常数数据段的开始。在与高级语言编写的程序相结合时，为了遵守高级语言的某些约定而需要特殊说明，这两条伪指令才使用。

（5）.CODE 定义代码段的开始。

（6）END 伪指令指出源程序文件到此为止，其后语句不予汇编，因此，一个程序文件有且只有一个，并通知连接程序该程序是从 END 后标明的“启动地址（这里是 START）”开始执行。“启动地址”可以是程序中的标号或过程名。若代码段的第一条语句便是程序要求的第一条可执行语句，则 END 后可默认指向这一标号。

【例 5.2】 在屏幕上输出“Hello World!”的简化段定义格式的汇编语言源程序。

程序如下：

```
.MODEL   SMALL
.STACK   100H
.DATA
NOTE     DB      'Hello world!',0ah,0dh,'$'
.CODE
BEGIN:   MOV     AX,@DATA
         MOV     DS,AX
         MOV     DX,OFFSET NOTE
         MOV     AH,09H
         INT     21H
         MOV     AH,4CH
         INT     21H
         END     BEGIN
```

说明：

（1）例 5.2 中.MODEL SMALL 指定该汇编程序所使用的内存模式为 SMALL。代码在一个 64KB 段中，数据在另一个 64KB 段中。

（2）例 5.2 中.STACK 100H 定义堆栈段为 100 字节长。

（3）例 5.2 中 END BEGIN 告诉汇编源程序到此结束，并通知连接程序该程序是从 END 后标明的 BEGIN 开始执行。此例中 BEGIN 代表“MOV AX,@DATA”语句的偏移地址。

5.1.3 完整段定义格式

完整段定义是一种标准的汇编语言书写形式，也是传统的书写形式，可使程序设计者对汇编程序实行全面控制，它适用于所有版本的汇编语言，既适用于将要生成的 EXE 文件对应的源程序的格式，也适用于将要生成的 COM 文件对应的源程序的格式。完整段定

义格式如下。

格式：

```
STACK   SEGMENT                              ; 定义堆栈段 STACK
        …                                    ; 分配堆栈段的大小
STACK   ENDS                                 ; 堆栈段结束
DATA    SEGMENT                              ; 定义数据段 DATA
        …                                    ; 定义数据
DATA    ENDS                                 ; 数据段结束
CODE    SEGMENT   'CODE'                     ; 定义代码段
        ASSUME CS :CODE,DS:DATA,SS:STACK     ; 确定 CS/DS/SS 指向的逻辑段
START:  MOV AX,DATA
        MOV     DS,AX                        ; 装填数据段寄存器
        …                                    ; 程序代码
        MOV     AX,4C00H                     ; 结束程序，返回 DOS 功能调用
        INT     21H
CODE    ENDS                                 ; 代码段结束
        …                                    ; 子程序代码
        END     START                        ; 汇编结束，程序起始点为 START
```

其中，

（1）SEGMENT/ENDS 定义逻辑段伪指令。为任何一个逻辑段命名，并指出以该段名命名的逻辑段的起止。其中段名是由用户自己给出的，每一对 SEGMENT/ENDS 前的段名必须保持一致，通常都写成 CODE、DATA、STACK 等。如果用户不写段名，则汇编时将自动为其加上一个段名：??SEG。

（2）ASSUME 定义寻址关系伪指令。出现在代码段中可执行的汇编指令之前，用以说明相应的逻辑段应通过相应的段寄存器寻址（包括变量与标号）。

任何一个源程序至少有一个逻辑代码段和一条作为源程序文件结束的伪指令 END，逻辑代码段用来放置指令性语句构成的程序；逻辑数据段根据程序本身功能的要求可有可无，也可以有多个；逻辑堆栈段可有可无，也可有多个，如果没有，那么连接时将产生一个警告性的错误，这对于没有堆栈操作的用户程序来说并不是错误，但如果用户程序用到堆栈的话，则最好设置自己的逻辑堆栈段。

【例 5.3】　在屏幕上输出“Hello World!”的完整段定义格式的汇编语言源程序。

程序如下：

```
DATA    SEGMENT                          ; 定义数据段
NOTE    DB      'Hello World!',0ah,0dh,'$'
DATA    ENDS                             ; 数据段结束
STACKl  SEGMENT  STACK                   ; 定义堆栈段
        DB      40H DUP(0)               ; 设置堆栈大小为 40H 字节
STACKl  ENDS
CODE    SEGMENT                          ; 定义代码段
        ASSUME  CS:CODE, DS:DATA ; 将代码段指定给 CS 寄存器，将数据段指定给 DS 寄存器
BEGIN:  MOV     AX,DATA                  ; AX 累加器
        MOV     DS,AX                    ; DS 数据段寄存器
        MOV     DX,OFFSET NOTE   ; 这句可以用 LEA DX,NOTE 来替代，DX 是数据通用寄存器
```

```
        MOV     AH,09H          ; 09H 对应的是显示字符串
        INT     21H             ; INT 21H 是用来调用功能的，具体区别用 AH 来识别
        MOV     AH,4CH          ; 4CH 功能在于让你退出程序
        INT     21H             ; 调用功能，由上面的 AH 决定，这句就是要退出程序了
CODE    ENDS                    ; 代码段结束
        END     BEGIN
```

说明：

（1）例 5.3 中 ASSUME 定义寻址关系伪指令，将代码段指定给 CS 寄存器，将数据段指定给 DS 寄存器。

（2）例 5.3 中分配堆栈段的大小为 40H 字节。

5.2 汇编语言数据

在源程序语句格式的 4 个组成部分中，操作数是指令的操作对象，操作参数之间用逗号分隔。操作数根据指令不同可以没有，可以有 1 个、2 个或多个。汇编语言程序中，指令操作数有数值型和地址型，数值型主要形式是常数和数值表达式；地址型主要形式是标号和名字（变量名、段名、过程名等）。硬指令的操作数有立即数、寄存器和存储单元。其中，立即数要用数值型参数表达，存储单元应该用地址型参数（存储器操作数）表达。

5.2.1 常量

常量也称为常数，表示一个固定的数值，在汇编期间，它的值完全确定，在程序运行期间也不会发生改变。它又分成多种形式。

1. 十进制常量

由 0～9 数字组成，以字母 D 或 d 结尾。默认情况下，后缀 D 可以省略。

例如：

```
101，249D
```

汇编语言对大小写不敏感，D 也可以使用 d 来代替。

2. 十六进制常量

由 0～9，A～F 组成，以字母 H 或 h 结尾。以字母 A～F 开头的十六进制数，前面要用 0 表达，以避免与其他符号混淆。

例如：

```
84H，0FFh，0A850H
```

3. 二进制常量

由 0 或 1 两个数字组成，以字母 B 或 b 结尾。

例如：

```
0l1010100B
```

4. 八进制常量

由 0～7 数字组成，以字母 Q 或 q 结尾。

例如：

```
234Q
```

各种进制的数据以后缀字母区分，默认不加后缀字母的是十进制数。

5. 字符串常量

用英文缩略号括起来的单个字符或多个字符，其数值是每个字符对应的ASCII码的值。

例如：

```
'A'，'ABC'，'Hello, World !'
```

这里需要注意的是，字符串常量中的字母是有大、小写区别的，即大写形式和小写形式表示不同的常量。

6. 实数常量

一个实数由整数、小数和指数 3 个部分组成，通常以十进制数形式表示。

格式：

```
±整数部分．小数部分 E±指数部分
```

其中，整数部分和小数部分合成的数值称为尾数，可以是一个带符号数，指数部分由字母 E 开头，它表示了尾数的数量级。

例如：

```
-243.876E-3
```

汇编期间，汇编程序把实数转换成 4 个字节、8 个字节或 10 个字节的二进制数形式存放在存储器中。因此实数必须由伪指令 DD、DQ、DT 定义，但实数不能出现在表达式中。

另外，实数也可以用十六进制数直接说明其二进制数编码形式，但是这个十六进制数必须以 0～9 开头，不带符号，并在最后用实数标识符 R 表示。

7. 符号常量

利用一个标识符表达的一个数值。常数若使用有意义的符号名来表示，可以提高程序的可读性，同时更具有通用性。MASM 提供等价机制，用来为常量定义符号名，符号定义伪指令有等价“EQU”和等号“=”伪指令。

格式：

```
符号名    EQU     数值表达式
符号名    EQU     <字符串>
符号名    =       数值表达式
```

其中，符号名的命名要有意义，EQU 为等价伪指令，=为等号伪指令。

EQU 可以将符号名定义成一个数值，或一个字符串，这个字符串甚至可以为一条处理器指令。

例如：

```
WriteAHChar     EQU     14
CarriageReturn  =       15
CallDOS         EQU     <INT 21H>
```

应用上述符号定义，下列左边的程序段可以写成右侧的等价形式：

```
MOV     AH,14           ; MOV     AH, WriteAHChar
MOV     DL,15           ; MOV     DL, CarriageReturn
INT     21H             ; CallDOS
```

EQU 用于数值等价时不能重复定义符号名，但=允许重复赋值。

例如：

```
X=7                    ; 同样 X EQU 7 是正确的
X=X+5                  ; 但是 X EQU X+5 是错误的
```

另外，MASM 6.0 还引入了 EQUTEXT 伪指令，只用于定义字符串常量。

【例 5.4】 常量及常量的使用。

程序如下：

```
DATA    SEGMENT                        ; 定义数据段
DATA1   EQU     31                     ; 加数 1
DATA2   EQU     22                     ; 加数 2
CALLDOS EQU     <INT 21H>
DATA    ENDS                           ; 数据段结束
CODE    SEGMENT                        ; 定义代码段
        ASSUME  CS:CODE, DS:DATA, ; 将代码段指定给 CS 寄存器，将数据段指定给 DS 寄存器
BEGIN:  MOV     AX,DATA                ; AX 累加器
        MOV     DS,AX                  ; DS 数据段寄存器
        MOV     AL,DATA1               ; 将 31 送入 AL 中
        MOV     BL,DATA2               ; 将 22 送入 BL 中
        ADD     AL,BL                  ; 将两数相加并存放在 AL 中
        MOV     AH,4CH                 ; 4CH 功能调用，返回 DOS 系统
        CALLDOS
CODE    ENDS                           ; 代码段结束
        END     BEGIN
```

说明：

在汇编语言源程序中，常量主要应用在以下场合。

（1）在指令语句的源操作数中作立即数。它可以是 8 位的，也可以是 16 位的，但必须与目的操作数的位数相一致。例如：

```
MOV     AX, 0ABCDH
ADD     CL, 12H
```

（2）在指令语句的基址/变址寻址方式、基址变址寻址方式中作位移量。例如：

```
MOV     BX,10H [BP]
MOV     20[SI],AX
SUB     DL,[BX][DI]
ADD     CL,12H[BX][SI]
```

（3）在数据定义伪指令（详见第 4.4.2 节）中，例如：

```
DB      12H             ; 定义一个字节数据
DW      1234H           ; 定义一个字数据
DD      12345678H       ; 定义一个双字数据
DB      'ABCD'          ; 定义四个字节的字符串数据
```

5.2.2 变量

变量是在程序运行期间可以随时改变的量。变量定义伪指令，为变量申请固定长度的

存储空间，并可以同时将相应的存储单元初始化。该类伪指令也是最常使用的伪指令。

格式：

```
变量名　伪指令　初值表
```

其中，

（1）变量名为用户自定义标识符，它是一个数据存储单元的名字，表示初值表首元素的逻辑地址，即用这个符号表示地址，常称为符号地址，而该存储单元中的数据是变量的值。变量名可以没有，这种情况下，汇编程序将直接为初值表分配空间，无符号地址。设置变量名是为了方便存取所指示的存储单元。为了提高程序的可读性和可理解性，建议用有意义的字符串给变量取名。

（2）初值表是用逗号分隔的参数，主要由数值常数、表达式或？、DUP 组成。其中？表示初值不确定，即未赋初值；重复初值可以用 DUP 进行定义。

（3）根据申请的主存空间单位，变量定义伪指令有 DB（定义字节变量）、DW（定义字变量）、DD（定义双字变量）、DQ（定义四字变量）、DT（定义五字变量）几种，下面做详细介绍。

1. 定义字节单元伪指令

字节定义伪指令 DB，用于分配一个字节或多个字节单元，并可以将它们初始化为指定值。初值表中每个数据一定是字节量（Byte），可以是 0～255 的无符号数或是−128～+127 带符号数，也可以是字符串常数。

例如：

```
DATA    SEGMENT                    ; 数据段开始
DX      DB      'D',6,0DH
        DB      2 DUP(100), ?
DY      DB      'BCD'
DATA    ENDS                       ; 数据段结束
```

上例中语句汇编后在存储器中的分配情况如图 5.2 所示，图中数据用十六进制表示。由单引号括起来的字符在内存中存放其 ASCII 码值。所以，'D'和 0Dh 是不同的，前者是字符'D'，后者是数值为 13 的十六进制编码。其中 2 DUP(100)定义 2 个字节数，为 100；而？定义的字节数据没有确定，所以图中用一横线表示。

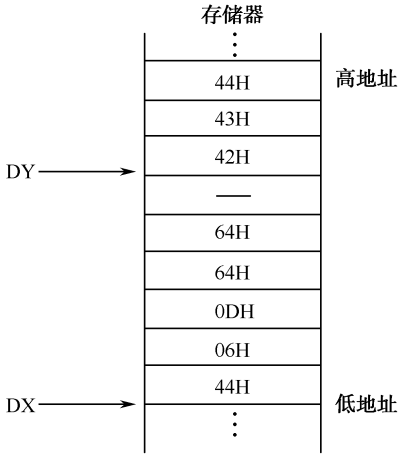


图 5.2　数据定义的存储形式

DUP 是定义重复数据操作符，其格式如下：

格式：

```
[变量名]　　数据定义伪指令　nDUP（表达式）
```

其中，n 为重复的次数，括号内的表达式是重复的内容。

例如：

```
DATAl    DB   10H DUP(?)
DATA2    DB   10H DUP('ABCD')
DATA3    DW   20 DUP (5)
DATA4    DB   10H DUP(4 DUP (1), 2, 3)
```

第一条语句表示分配了 10H 个字节，且每个字节预置不定内容。第二条语句表示分配了 10H 个字符串'ABCD'，共占 10H×4=40H 个字节。第三条语句表示分配了 20 个字单元，每个字单元预置内容为 5，共占 20×2×1=20 个字节。第四条语句的 DUP 是嵌套使用的。最外层 DUP 后的括号表达式等效为（1，1，1，1，2，3），表示分配了 10H 个数据序列（1，1，1，1，2，3），共占 10H×6=60H 个字节。

伪指令 DB 还可以为字符串的每个字符分配一个字节单元，必须用成对的单引号或双引号把所有的字符括起来，字符串必须用单引号括起来，且不超过 255 个字符。字符串自左向右以字符的 ASCII 编码按地址递增的排列顺序依次存放。

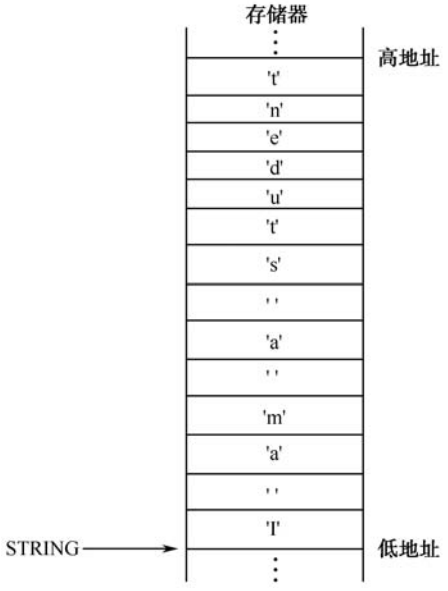


图 5.3　变量 STRING 在存储器中存放情况

例如：

```
STRING    DB   'I am a student'
```

STRING 在存储器中的存放情况如图 5.3 所示。这里要注意的是，为了看起来方便，它并没有用字符的 ASCII 码来存放在相应的存储单元内，而直接用该字符，请不要引起误解。也可改写为另一种等价的语句：

```
STRING    DB   'I', ' ', 'a', 'm', ' ', 'a', ' ', 's', 't', 'u', 'd', 'e', 'n', 't'
```

显然，前者的说明要比后者方便得多，所以，在程序中都采用前者的书写方式。

不带引号的？表示可预置任何内容。

例如：

```
DATA_B    DB   ?, ?
```

上例中语句要求汇编程序分配了两个字节单元，并且这些单元的内容可以是任意值。

2. 定义字单元伪指令

字定义伪指令 DW，用于分配一个或多个字单元，并可以将它们初始化为指定值。初值表中每个数据一定是字量，一个字单元可用于存放任何 16 位数据，如一个段地址、一个偏移地址、两个字符、0～65535 之间的无符号数或−32768～+32767 之间的带符号数。

例如：

```
DATA      SEGMNET                ; 数据段开始
count     DW       80h,?,'AB'
maxnum    EQU      64h
number    DW       maxnum
DATA      ENDS                   ; 数据段结束
```

上述语句汇编后在存储器中的分配情况如图 5.4 所示。由于字变量的数据是按照“高高低低”的原则存放在存储单元中的，而字节数据是按照排列顺序存放在存储单元中的，所以'AB'存放为 4142H；符号常量 maxnum 等于 64H，它并不占用存储空间。

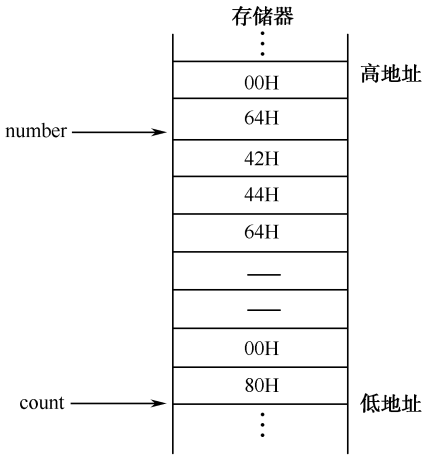


图 5.4　数据定义的存储形式

在前面介绍的存储器寻址方式中，提到位移量可以用符号表示，现在补充定义如下：

```
WNUM    EQU 5678H              ; 定义 WNUM 为常量
COUNT   DW  20H                ; 定义 COUNT 为变量，假设它在数据段的偏移地址为 10H
```

这样，以下各个指令的功能见注释部分：

```
MOV     AX,[BX+SI+WNUM]        ; 等价于 MOV AX,[BX+SI+5678H]
MOV     AX,COUNT               ; 等价于 MOV AX,[0010H]
MOV     AX,[SI+COUNT]          ; 相同于 MOV AX,COUNT[SI]
                               ; 等价于 MOV AX,[SI+10H]
LEA     BX, COUNT              ; 等价于 LEA BX,[0010H]
MOV     BX, OFFSET COUNT       ; 于 MOV BX,0010H
```

从上面的指令看到，变量实际上表达的是主存地址，而且具有属性。仅使用变量名或者再加一个常量是直接寻址，再加上一个寄存器是寄存器相对寻址，加上两个寄存器则是基址变址相对寻址方式。利用后面介绍的 OFFSET 操作符可以获得这个变量的偏移地址。

注意：

DW 最多只能为两个字符组成的字符串分配存储单元，并且按照字符自左向右的排列顺序，第一个字符的 ASCII 码存放在字单元的高字节，第二个字符的 ASCII 码存放在字单元的低字节。

3. 定义双字单元伪指令

双字定义伪指令 DD，用于分配一个或多个双字单元，并可以将它们初始化为指定值。初值表中每个数据是一个 32 位的双字量，可以是有符号或无符号的 32 位整数，也可以用来表达 16 位段地址（高位字）和 16 位偏移地址（低位字）的远指针。

例如：

```
DATA      SEGMENT                          ; 数据段开始
DATADD1     DD   27 ?,12345678H
DATADD2     DD   05606008H
DATADD3     DD   'AB','CD'
DATA      ENDS                             ; 数据段结束
```

注意：

伪指令 DD 最多只能为两个字符组成的字符串分配存储单元，而且按照字符自左向右的排列顺序，第一个字符的 ASCII 码存放在第一个字单元的高字节，第二个字符的 ASCII 码存放在第一个字单元的低字节，第二个字单元的高、低字节内容都存放 00H。

4. 其他数据定义伪指令

（1）定义 3 字伪指令 DF。DF 用于为一个或多个 6 字节变量分配空间及初始化。6 字节常用在 32 位 CPU 中，表示一个 48 位远指针（16 位段选择器：32 位偏移地址）。

（2）定义 4 字伪指令 DQ。DQ 用于为一个或多个 8 字节变量分配空间及初始化。8 字节变量可以表示一个 6 4 位整数。

（3）定义 10 字节伪指令 DT。DT 用于为一个或多个 10 字节变量分配空间及初始化。从 MASM 6.0 开始，变量定义伪指令 DB、DW、DD、DF、DQ、DT 被建议使用新的表达形式，依次为 BYTE、WORD、DWORD、FWORD、QWORD、TBYTE，对应的两指令功能相同，前者只是为了与旧版本兼容而被保留下来。另外，还有 SBYTE、SWORD、SDWORD 指令也用于定义字节、字和双字单元的存储空间，但是，它们专门用于带符号数的定义和初始化。

【例 5.5】 变量的定义。

代码段如下：

```
DATA    SEGMENT                          ; 定义数据段
        DATAl   DB      11H
        DATA2   DW      DATA1
        DATA3   DD      DATA2
        DATA4   DB      2 DUP('B'),5
DATA    ENDS                             ; 数据段结束
```

说明：

（1）例 5.5 中逻辑段 DATA 在存储器中的存放情况如图 5.5 所示。这一片连续的数据存储单元形成一个存储区，其类型由它们前面的数据定义伪指令确定，各存储单元的初值由各表达式的值顺序填入。

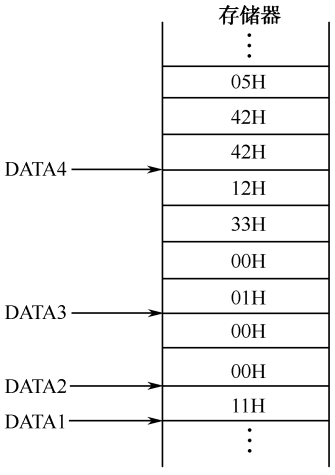


图 5.5 逻辑段 DATA 在存储器中的存放情况

（2）如果地址表达式为变量名或是标号，那么用伪指令 DW 定义，则是用它的偏移量来初始化变量；用伪指令 DD 定义，则是用它的段基值和偏移量来初始化变量，且段基值存放在高字单元，偏移量存放在低字单元。

例 5.5 中逻辑段 DATA 中定义了 4 个变量 DATA1、DATA2、DATA3 和 DATA4。其中，DATA1 的内容为 11H；DATA2 的内容为 DATA1 在逻辑段 DATA 中的偏移量，即为 00H；DATA3 的低字内容为 DATA2 在逻辑段 DATA 中的偏移量，即为 01H；DATA3 的高字内容为 DATA2 所在逻辑段 DATA 的段基值，假设为 1233H；DATA4 的内容为 42H、42H 和 05H。

注意：

地址表达式前面的数据定义伪指令只能是 DW 和 DD。

5.2.3　标号

标号是一种特殊的标识符，它代表代码段中的某个具体位置，主要用于表明转移的目标位置。常作为转移指令或子程序调用指令的目的操作数。其格式如下。

格式：

```
标号: 汇编语言指令          ;注释
```

其中，标号必须是一个合法的标识符，在其后面紧跟一个冒号“:”，冒号与汇编语言指令之间要有分隔符。通常用若干个空格、TAB 来作分隔符，一般用分隔符使有关内容对齐为宜。

例如：

```
            …
L1:         …
            LOOP    L1
            …
            JNE     NEXT
            …
NEXT:       …
            …
```

5.2.4　变量和标号的属性

变量和标号都是一个符号地址，变量值会根据其数据类型与该地址开始的若干个存储单元中所存的数值相对应，而标号所对应的存储单元中存放的是指令代码。虽然它们在某些性质上有所不同，但它们都代表一个存储单元的地址，所以，它们都具有存储单元的属性。除此之外，它们还有各自特殊的属性。

下面介绍变量和标号的属性及其有关操作符。

1. 段属性操作符

段属性操作符（SEG）返回该标识符所在段的段地址。一般情况下只会取变量所在段的段地址，而很少取标号所在段的段地址。格式如下。

格式：

```
SEG  变量名/标号
```

例如：

```
            …
DATA1   DW   80H
DATA2   DB   10 DUP(9)                ; 数据段的变量定义
            …
            MOV     AX, SEG DATA1     ; 代码段的指令
            MOV     BX, SEG DATA2
```

由于 DATA1 和 DATA2 在同一段中定义，所以，寄存器 AX 和 BX 的值是相等的。

2. 偏移量属性操作符

偏移量属性操作符（OFFSET）返回该标识符离它所在段的段地址的字节数。一般情况，只会取内存变量的偏移量，而不太关心标号的偏移量。格式如下。

格式：

```
OFFSET 变量名/标号
```

例如：

```
DATA1   DD   12345678H, 86                    ; 数据段的变量定义
DATA2   DW   ?, 12H
DATA3   DB   10 DUP(9)
        …
        MOV      AX, OFFSET DATA2            ; 代码段的指令
        MOV      BX, OFFSET DATA3
        …
```

假设 DATA1 是数据段的第一个被定义的变量名，则它的偏移量为 0，DATA2 的偏移量为 8，因为它要跳过两个双字，所以 DATA3 的偏移量为 12，因此，BX 的值为 12，AX 的值为 8。

程序设计中，还经常利用 OFFSET 计算一段数据或代码的长度（字节数）。例如，主程序开始有一个 START 标号，最后一个指令后有一个 DONE 标号，则 OFFSET DONE-OFFSET START 将得到主程序以字节为单位的长度。

3. 类型属性操作符

类型属性操作符（TYPE）是返回该变量所占字节数，或标号的“远”（FAR）、“近”（NEAR）类型。常用标识符的类型值如表 5.4 所示。格式如下。

格式：

```
TYPE 变量名/标号
```

表 5.4 常用标识符的类型值列表

标识符种类	字节变量（DB）	字变量（DW）	双字变量（DD）	近标号（NEAR）	远标号（FAR）
TYPE 的值	1	2	4	−1	−2

例如：

```
DATA1   DW  1          ; 数据段的变量定义
DATA2   DB   10 DUP(?)
…
MOV     AL,TYPE DATA1
MOV     BL,TYPE DATA2
…
```

按属性 TYPE 的含义，AL 和 BL 的值分别为 2 和 1。

在汇编语言程序设计中，可以根据 TYPE 运算符的返回值来修改地址指针。

例如：

```
ARRAY   DW       10H DUP (0)
        …
```

```
        MOV     SI,OFFSET  ARRAY
LOOP1:  MOV     AX, [SI]
        …                                   ; 对数组逐个元素进行某种操作
        ADD     SI,TYPE  ARRAY              ; 修改地址指针
        LOOP    LOOP1
        …
```

4. 长度属性操作符

长度属性操作符（LENGTH）是针对变量的操作符，它返回重复操作符 DUP 中的重复数。如果有嵌套的 DUP，则只返回最外层的重复数；如果没有操作符 DUP，返回的值总是 1。格式如下所示。

格式：

```
LENGTH 变量名
```

例如：

```
DATA1   DB      20H DUP (0)
DATA2   DB      10H,20H,30H,40H
DATA3   DW      30H DUP (0,1,20 DUP(2))
DATA4   DB      'ABCDEFG'
        …
        MOV     AL, LENGTH  DATAl     ; (AL)=20H
        MOV     BL, LENGTH  DATA2     ; (BL)=1
        MOV     CX, LENGTH  DATA3     ; (CX)=30H
        MOV     DX, LENGTH  DATA4     ; (DX)=1
        …
```

根据属性 LENGTH 的含义，AL、BL、CX 和 DX 的值分别为 20H、1、30H 和 1。

在汇编语言程序设计中，可以根据 LENGTH 运算符的返回值得到数组的元素个数，可以应用在循环程序中。

例如：

```
ARRAY   DW      20H DUP (12)
        …
        MOV     CX,LENGTH  ARRAY      ; 确定数组 ARRAY 需要处理的元素个数
LOOP1:  …                             ; 对数组逐个元素进行某种操作
        LOOP    LOOP1
        …
```

5. 容量属性操作符

容量属性操作符（SIZE）也仅仅是针对变量的操作符，得到的运算结果是数组变量所占的总字节数，即返回值等于 LENGTH 和 TYPE 两个运算符返回值的乘积。它的返回值按下列公式计算。

公式：

SIZE 变量=(LENGTH 变量)×(TYPE 变量)

例如：

```
DATA1   DB      20H DUP (0)
DATA2   DB      10H,20H,30H,40H
DATA3   DW      30H DUP (0,1,20 DUP(2))
```

```
DATA4   DB      'ABCDEFG'
        …
        MOV     AL, SIZE   DATAl    ; (AL)=20H*1=20H
        MOV     BL, SIZE   DATA2    ; (BL)=1*1=1
        MOV     CX, SIZE   DATA3    ; (CX)=30H*2=60H
        MOV     DX, SIZE   DATA4    ; (DX)=1*1=1
        …
```

根据属性 SIZE 属性含义，AL、BL、CX 和 DX 的值分别为 20H、1、60H 和 1。

6. 强制属性操作符

在汇编程序中，有时要以不同的属性来访问同一个存储单元，或对一些不确定的存储属性显式指定等，这时，就需要强制属性操作符 PTR。假设有如下这样一条指令：

```
MOV     [BX], 1H
```

其目标操作数[BX]是寄存器间接寻址方式，它指向一个存储单元。在作传送操作时，是把“1H”扩展成 8 位作字节传送，还是扩展成 16 位作字传送呢？因为[BX]指向的存储单元可以是字节或字的首地址，这就使该指令具有两个含义。含有该指令的程序在汇编时，可能会产生警告或出错信息。为了使指令中存储单元操作数具有明确的属性，可以使用强制属性操作符 PTR。其一般格式如下：

格式：

```
数据类型   PTR   地址表达式
```

其中，数据类型是前面所学的各种数据类型，常用的数据类型有 BYTE、WORD、DWORD、NEAR 和 FAR 等。

为了明确指令中存储单元的属性，指令“MOV [BX], 1H”可改写成如下两种形式：

```
MOV   BYTE PTR [BX], 1H      或      MOV   WORD PTR [BX], 1H
```

注意：

PTR 这种修改或指定是临时的，仅在有此运算符的语句内有效。

在指令中用操作符 PTR 强制后，不管其后的地址表达式原数据类型是什么，在本指令中就以 PTR 前面的类型为准。该强制属性只在本指令有效，是一种临时性的属性，它不会改变原内存单元的定义属性。

例如：

```
DATA1   DW      1234H, 5678H
DATA2   DB      5,7
…
        MOV     AX,WORD PTR DATA2
        MOV     BH,BYTE PTR DATA1
…
```

上例中第一条指令把 DATA2 开始的两个字节拼接成一个字，执行指令后，AX 为 0705H。第二条指令把字 DATA1 的低字节传送给 BH，执行指令后，BH 为 34H。

7. 存储单元别名操作符

由上一节的内容可以知道，在程序中，如果需要以另一种数据类型访问某一存储单元，可用强制属性操作符 PTR 实现。但如果在程序中要经常以某种其他的数据类型来访问该存

储单元的话，那么，就必须在每次访问时都要加上强制属性操作符 PTR。这样做虽然可行，但在编写程序时就显得比较麻烦。

为了方便，汇编语言提供了另一种操作符 THIS，它为同一存储单元取另一别名，该别名可具有其自身的数据属性，但其段地址和偏移量不变。操作符 THIS 的一般格式如下。

格式：

```
THIS 数据类型
```

其中，数据类型是前面所学的各种数据类型，常用的数据类型有 BYTE、WORD、DWORD、NEAR 和 FAR 等。

例如：

```
WBUFFER EQU     THIS WORD
BUFFER          DB      20 DUP(?)
```

这样就给同一个存储单元取了两个具有不同数据类型的变量名。于是，在指令中，引用不同的变量名，就使用其不同的数据属性：

（1）如果引用变量名 WBUFFER，则是按“字”属性来访问；

（2）如果引用变量名 BUFFER，则是按“字节”属性来访问。

如此一来，指令“MOV AX,WORD PTR BUFFER”和“MOV AX, WBUFFER”是等效的，所不同的是，当以“字”属性访问 BUFFER 存储区时，不必使用强制属性符 PTR，而改用“字”属性变量 WBUFFER 即可。

5.3　表达式与运算符

表达式是程序设计课程里一个重要的基本概念，它是由运算符、操作符、括号、常量和一些符号连在一起的式子。在汇编语言中，表达式可以分为地址表达式和数值表达式。

地址表达式是计算存储单元地址的表达式，它可以由标号、变量和由括号括起来的基址或变址寄存器组成。其计算结果表示一个存储单元的地址，而不是该存储单元的值。

数值表达式是在汇编过程中能够由汇编程序计算其值的表达式，其组成部分在汇编时就能完全确定，它通常是一些常量的运算组合。下面着重介绍运算符。

5.3.1　算术运算符

算术运算符包括符号：+（正）、-（负），运算符：+（加）、-（减）、*（乘）、/（除）和 MOD（取模）。这些运算符和常量、括号可组成数值表达式。算术运算符的语法格式和操作说明如表 5.5 所示。其中表中的前两个是单项运算符，仅表示数的正负。+、-、*、/运算符用于对数据进行加、减、乘和除，参加运算的数和运算的结果都是整数，除法运算的结果只取商的整数部分。而 MOD 运算符进行整数除法，运算结果只取商的余数部分。[]运算符为求下标。

例如：

```
MOV     AL,-9
MOV     BX, 11*4+20/2
ADD     CX, 5684H MOD2
```

汇编结果是：

```
MOV     AL,-9
MOV     BX, 84
ADD     CX, 0
```

表 5.5　算术运算符

运算符	语法格式	操作说明
+	+ 表达式	正数（单项运算符）
-	- 表达式	负数（单项运算符）
+	表达 1 + 表达式 2	加法
-	表达式 1 - 表达式 2	减法
*	表达式 1 * 表达式 2	乘法
/	表达式 1 / 表达式 2	除法
MOD	表达式 1 MOD 表达式 2	取余
[ ]	表达式 1[表达式 2]	求下标

减法运算符还可以用于计算同段内两个操作数地址（以变量表示）的之间的距离（字节数）。

例如：

```
DATAl     DB   78H,56H
DATA2     DB   'AB'
DATA3     =    DATA2 - DATA1
```

上例中 DATA3 是两个变量的偏移量相减，其结果正是以 DATA1 为首址占用的 2 个字节单元数。下标运算符“[]”用来对数组元素进行存取操作，表示加法，即把表达式 1 和表达式 2 相加后形成一个存储器操作数地址。

例如：

```
MOV AX, DATA[10H]       ;等价于 MOV    AX, DATA+10H
```

即源操作数是以 DATA 为首址的偏移 10H、11H 两个字节组成的一个字。[10H]是一个数组的下标，也可以用寄存器来存放下标。

例如：

```
MOV     AX, ARRAY [BX] [SI]
MOV     AX, ARRAY[BX+ SI]
MOV     AX, [ARRAY+BX] [SI]
MOV     AX, [ARRAY+ SI] [BX]
MOV     AX, [ARRAY+BX+ SI]
```

上例中 5 条指令语句都是等价的。

注意：

当算术运算对象为地址操作数时，应保证其结果是一个有意义的存储器地址，因而通常只使用+、-运算。

两个存储器操作数地址（如变量）不能相加，下面几个指令语句都是错误的：

```
MOV     AX, ARRAY+BX+ SI
```

```
MOV     AX, ARRAY+BX [SI]
MOV     AX, ARRAY + DATA
```

其中，ARRAY 和 DATA 是变量。

5.3.2　移位运算符

移位运算符用于数据的逻辑移位运算，有左移（SHL）和右移（SHR），运算对象也必须是数值型的操作数。移位运算符的语法格式和操作说明如表 5.6 所示。移位的次数由运算符 SHR 和 SHL 右边的次数确定，如果次数≥16，则结果恒为 0。

例如：

```
MOV     AL,4CH SHL 2
MOV     AH,4CH SHR 1
```

汇编的结果是：

```
MOV     AL,00110000B
MOV     AH,00100110B
```

表 5.6　移位运算符

运算符	语法格式	操作
SHR	表达式 SHR 次数	右移（最左边用 0 补齐）
SHL	表达式 SHL 次数	左移（最右边用 0 补齐）

注意：

移位运算符与移位指令是完全不同的概念。移位运算符是对某一个具体的数据（常数）在汇编时完成移位的，而移位指令是对一个寄存器或存储单元内容在程序运行时执行移位的。

5.3.3　逻辑运算符

逻辑运算符包括非（NOT）、与（AND）、或（OR）和异或（XOR），用于数据的按位逻辑运算，逻辑运算符的语法格式和操作说明如表 5.7 所示。逻辑运算符的运算对象必须是数值型的操作数，并且按位运算。

例如：

```
MOV     BL,NOT 0FFH
MOV     CL,4CH AND 0F0H
MOV     CH,09H OR 30H
MOV     AL,4CH XOR 0FFH
```

汇编的结果是：

```
MOV     BL,0
MOV     CL,40H
MOV     CH,39H
MOV     AL,0B3H
```

表 5.7 逻辑运算符

运 算 符	语法格式	操 作
NOT	NOT 表达式	逻辑非
AND	表达式 1 AND 表达式 2	逻辑与
OR	表达式 1 OR 表达式 2	逻辑或
XOR	表达式 1 XOR 表达式 2	逻辑异或

注意：

逻辑运算符与逻辑运算指令是完全不同的两回事，前者是在汇编时完成的，后者是在运行时执行的。

5.3.4 关系运算符

关系运算符包括相等（EQ）、不等（NE）、小于（LT）、不大于（LE）、大于（GT）和不小于（GE），用于比较两个表达式的大小。表达式一定是常数或同段的变量。如果是常数，那么按无符号数进行比较；如果是变量，那么比较它们的偏移量。比较结果以真（全 1）或假（全 0）的形式给出。关系运算符的语法格式和操作说明如表 5.8 所示。

例如：

```
MOV     AL,4CH EQ 2
MOV     AH,4CH GT 1
```

汇编的结果是：

```
MOV     AL,0
MOV     AH,0FFH
```

假设某数据段有：

```
DATA1   DW    NUM  LT  25H
```

那么在程序汇编时，如果 NUI<25H，那么变量 DATA1 的内容为 0FFFFH；否则它的内容为 0。

表 5.8 关系运算符

运 算 符	语法格式	操 作
EQ	表达式 1 EQ 表达式 2	两个表达式相等时为真
NE	表达式 1 NE 表达式 2	两个表达式不相等时为真
LT	表达式 1 LT 表达式 2	表达式 1<表达式 2 为真
LE	表达式 1 LE 表达式 2	表达式 1≤表达式 2 为真
GT	表达式 1 GT 表达式 2	表达式 1>表达式 2 为真
GE	表达式 1 GE 表达式 2	表达式 1≥表达式 2 为真

5.3.5 其他运算符

在汇编语言中，还有其他可在数值表达式中使用的操作符，如下所示。

（1）HIGH（高 8 位）、LOW（低 8 位）；

（2）SEG（段地址）、OFFSET（偏移量）；

（3）TYPE（标识符类型）、LENGTH（变量长度）、SIZE（变量容量）；

（4）WIDTH（记录/记录字段宽度）、MASK（记录/记录字段的屏蔽位）等。

HIGH 和 LOW 分别是选取表达式计算结果的高 8 位和低 8 位。其使用格式如下。

格式：

```
HIGH    表达式
LOW     表达式
```

例如：

```
HIGH    (1234H+100H)
LOW     1234H
```

上例中选取结果分别为 13H 和 34H。

5.3.6　运算符的优先级

当表达式中同时有几个运算符时，按运算符优先级顺序运算，运算符优先级顺序如表 5.9 所示，汇编源程序时按照以下规则计算表达式的值。

表 5.9　运算符优先级

优先级		运算符
高 ↓ 低	1	LENGTH，SIZE，WIDTH，MASK，( )，[ ]，< >
	2	PTR，OFFSET，SEG，TYPE，THIS
	3	HIGH，LOW
	4	+，－（单项运算符）
	5	*，/，MOD，SHL，SHR
	6	+，－（双项运算符）
	7	EQ，NE，LT，LE，GT，GE
	8	NOT
	9	AND
	10	OR，XOR

（1）先执行优先级高的运算；

（2）优先级相同的运算符则按从左到右的规则运算；

（3）可以用圆括号改变运算的顺序。

例如：

```
MOV     AL, 10  OR  5  AND  l
MOV     AH, (10  OR  5)  AND  1
```

汇编后为：

```
MOV     AL, 11
MOV     AH, 1
```

5.4 常用伪指令

伪指令又称为伪操作，它是程序源代码在汇编期间由汇编器识别和执行的指令，可以完成选择处理器、定义程序模式、定义数据、分配存储区、指定程序结束等功能，除了定义数据及分配存储区为指令分配存储器空间外，其他伪指令不生成目标码。伪指令是汇编语言的保留字，不能够用来命名变量名或标号。

8086/8088 宏汇编语言提供了多种伪指令，其中在前面也介绍了一些伪指令，如 SEGMENT/ENDS 段定义伪指令、等值伪指令“EQU”、等号伪指令“=”、DB 伪指令、DW 伪指令、DD 伪指令、DQ 伪指令和 DT 伪指令等，本节只介绍部分常用的伪指令，另外还有一些伪指令将在后续的章节中说明。

5.4.1 属性定义伪指令 LABEL

符号名定义伪指令 LABEL 的功能是为当前的变量或标号定义一个新的类型。与存储单元别名操作符 THIS 的功能类似，该伪指令定义一个指定的符号名，该符号名的段地址和段内偏移地址与下面紧跟着的一条变量定义语句中定义的变量的相应属性完全相同，但是该符号的类型是新指定的，伪指令 LABEL 的一般格式如下。

格式：

```
变量名/标号   LABEL   数据类型
```

其中，

（1）伪指令 LABEL 本身不开辟新的存储单元，但它可以改变变量或标号的类型属性，使这个变量或标号对于不同的引用可以具有不同的类型属性。

（2）数据类型就是常用的数据类型，如 BYTE、WORD、DWORD、NEAR、FAR 等。前面 3 种类型是变量的类型，后面 2 种类型是标号的类型。

伪指令 LABEL 的使用有以下两种情况。

（1）与指令语句连用。改变的是标号的属性，所以类型只能为 NEAR 和 FAR 两种。

（2）与数据定义伪指令语句连用。改变的是变量的属性，所以类型为 BYTE、WORD、DWORD 等。

例如：

```
DATA1    LABEL   BYTE
DATA2    DW   10H   DUP (1)
```

数据定义伪指令“DW 10H DUP(1)”所定义的存储单元由两个变量 DATA1 和 DATA2 所指定，此两个变量具有相同的段基值和偏移量，但是它们的类型属性不同，DATA1 是字节类型，对其存取是按字节进行的，而 DATA2 是字类型，对其存取是按字（两个字节）进行的。

例如：

```
MOV      AL, DATA1 + 4
```

上例中是把第 5 个字节（从 1 开始算）的内容送入 AL 中，如果想取出第 3 个字（从 1 开始算）的内容送入 AX 中，指令如下：

```
MOV      AX, DATA2+ 4
```

5.4.2　段寻址伪指令 ASSUME

汇编程序根据段开始语句和段结束语句判断出源程序的段划分，为了有效地产生目标代码，汇编程序还要了解各程序段与段寄存器间的对应关系。段寄存器与程序段间的对应关系由段使用设定语句说明。

段寻址伪指令 ASSUME 是用来指示汇编程序哪些段是当前段，以及这些段与段寄存器之间的对应关系。ASSUME 伪指令只影响汇编源程序的设定，而不影响程序运行时段寄存器的设定。段寻址伪指令的格式如下。

格式：

```
ASSUME 段寄存器名：段名，段寄存器名：段名，…
```

其中，

（1）伪指令 ASSUME 也可以用来取消或修改逻辑段与段寄存器之间的对应关系，它可以取消或修改一个或几个逻辑段与段寄存器之间的关系，也可以全部取消或修改。设置删除用关键字 NOTHING 完成。

（2）段寄存器名可以是 CS、DS、SS、ES，段名就是段开始语句和段结束语句中规定的段名，是用段定义伪指令 SEGMENT/ENDS 定义的，段寄存器名与段名之间必须用冒号“：”隔开。

例如：

```
DATA1   SEGMENT                          ; 定义一个数据段，段名为 DATA1
        VARW1   DW      12
        …
DATA1   ENDS
DATA2   SGEMNET                          ; 定义一个数据段，段名为 DATA2
        VARW2   DW      0
        VARW3   DW      0
DATA2   ENDS
CODE    SEGMENT                          ; 定义一个代码段，段名为 CODE
        ASSUME  CS:CODE,DS:DATA1,ES:DATA2
        MOV     AX,DATA1
        MOV     DS,AX
        MOV     AX,DATA2
        MOV     ES,AX
        …
        MOV     AX,VARW1
        MOV     VARW2,AX
        …
        ASSUME  DS:DATA2,ES:NOTHING
        MOV     AX,DATA2
        MOV     DS,AX
        …
        MOV     AX,VARW2
        MOV     VARW3,AX
```

```
        …
CODE    ENDS
```

上例中变量 VARW1 在 DATA1 中定义，变量 VARW2 和 VARW3 在 DATA2 中定义，第一条 ASSUME 伪指令告诉汇编程序，数据段寄存器 DS 对应 DATA1，附加段寄存器 ES 对应 DATA2，代码段寄存器 CS 对应 CODE。由于一般操作数寻址隐含使用的是 DS，因此汇编程序在汇编“MOV VARW2,AX”时，将自动加上段超越前缀，即成为“MOV ES:VARW2,AX”。而第二条 ASSUME 语句把 DS 与 DATA2 对应，而变量 VARW2 和 VARW3 均在 DATA2 段中，所以汇编程序在汇编“MOV AX,VARW2”和“MOV VARW3,AX”时就不再加上段超越前缀。假设没有第二条 ASSUME 语句，那么汇编程序也要为上述两条指令加上段超越前缀“ES：”。如果在第二条 ASSUME 语句后，安排指令“MOV AX,VARW1”，汇编程序将发出无法访问变量 VARW1 的错误信息，原因是第二条 ASSUME 语句解除了 DS 寄存器与 DATA1 段的对应关系，DATA1 段不再与任何段寄存器对应。如果删除全部 3 个段寄存器与逻辑段之间的关联设置，可以使用如下指令。

```
ASSUME  NOTHING
```

注意：

ASSUME 伪指令不产生任何机器目标代码，它仅仅告诉汇编程序各逻辑段与各段寄存器之间的对应关系，即各逻辑段使用段寄存器的情况，并没有将各逻辑段的段基值装入各段寄存器中。

如果没有用 ASSUME 伪指令中指定逻辑段与段寄存器之间的关系，那么在变量名或地址表达式前要用段前缀指明要使用的段寄存器，例如，MOV AX, ES:VARW。

5.4.3 过程定义伪指令 PROC/ENDP

利用过程定义伪指令 PROC/ENDP 语句，可把程序片段说明为具有近类型或远类型的（说明见下页）过程，并且能给过程起一个名字。此外在程序设计中，PROC/ENDP 伪指令常常把一个复杂问题分解成多个简单的子问题，构造成多个子程序，再把这些子程序搭建成整个软件，这样就形成了模块化程序设计的基本思想，有利于提高每个模块的内聚性，降低模块与模块之间的耦合性。对于程序中经常用到的一些功能模块，也可以设计成一个子程序，方便多次调用，减少很多重复代码。MASM 宏汇编程序用过程来构造子程序。过程定义伪指令的格式如下。

格式：

```
过程名   PROC  [NEAR/FAR]
         …
RET
         …
过程名   ENDP
```

其中，

（1）过程名的命名与普通标号的命名方法相同。像普通标号一样，过程名具有段值、偏移量和类型这三个属性。过程名的段值和偏移量是对应过程入口（过程定义开始伪指令语句后的指令语句）的段值和偏移量，过程名的类型就是过程的类型。开始语句 PROC 和

结束语句 ENDP 前的过程名称必须一致，从而保持配对。

（2）过程的类型在过程定义开始语句 PROC 中指定，过程可以被指定为近（NEAR）类型，也可以被指定为远（FAR）类型。如果不指定，则通常默认为近类型。使用段内调用指令还是段间调用指令，由被调用过程的类型决定。因此，如果一个过程要被别的段的程序调用，就应该把它说明为 FAR 类型。如果具有 NEAR 类型，那么该过程仅能被其所在段调用。

（3）每一个过程中一定有返回指令 RET，它可以在过程的任何位置，不一定非要放在过程的最后，另外，一个过程中也不一定只有一条返回指令，可以有多条返回指令，为过程设置多个出口，但是过程执行的最后一条指令一定是返回指令 RET。

例如：

```
CODE    SEGMENT
ASSUME  CS：CODE, …
…
PROCl   PROC    NEAR
…
RET
…
PROCl   ENDP
…
PROC2   PROC    FAR
…
RET
…
PROC2   ENDP
…
CODE    ENDS
```

上例中过程 PROC1 的类型属性为 NEAR，它仅能供本段（CODE 段）内的其他程序所调用。过程 PROC2 的类型属性为 FAR，它不仅能被本段内的程序调用，还可以供其他段（非 CODE 段）的程序调用。

当调用类型属性为 NEAR 时，CPU 仅把当前 IP 的内容（即调用程序断点的偏移量）压入堆栈，作为返回地址保存起来，然后把过程的偏移量送入 IP，RET 返回时，从堆栈中弹出返回地址并送入 IP。调用类型属性为 FAR 时，CPU 要把当前 CS 和 IP 的内容一起作为返回地址压入堆栈（共 4 个字节）；RET 返回时，从堆栈中弹出 4 个字节的内容分别送入 IP 和 CS。当 FAR 类型的过程被本段内程序调用时，CPU 仍按段间调用和返回的方式来处理段内调用和返回。

5.4.4　调整偏移量伪指令

用数据定义伪指令分配的数据是按顺序一个接一个存放在数据段中的。但有时，用户希望能够控制数据的偏移地址。例如，使数据对齐可以加快数据的存取速度。MASM 提供几个这样的伪指令。常用的调整偏移量伪指令有 ORG、EVEN 和 ALIGN，它们的主要目的是为了更有效地读取内存单元的内容。

1. 定位伪指令 ORG

ORG 伪指令使它后面的数据或指令从参数指定的地址开始，是将当前偏移地址指针指向参数表达的偏移地址。

格式：

```
ORG 参数
```

其中，

参数表达式是以 65536 为模进行计算的，并且它的值一定是正数。参数中可以包括当前位置计数器$。

例如：

```
DATA    SEGMENT
        ORG     200H
V1      DW      1,2                 ; V1 的偏移地址为 200H
        ORG     $ + 10H             ; 跳过 10H 个字节
V2      DB      ?                   ; V2 的偏移地址为 214H
DATA    ENDS
```

上例中第一条 ORG 语句“ORG 200H”的作用是使得在逻辑段 DATA 中变量 V1 的段内偏移量为 200H，而不是 0H。第二条 ORG 语句“ORG $+10H”表示把当前位置计算器的值加上 10H，其作用是保留了 10H 个字节的存储单元，因此变量 V2 的段内偏移量为 214H。

汇编语言程序中，符号“$”表示当前偏移地址值。例如，在偏移地址 100H 单元开始定义“DW 1,2,$+4,$+4”，那么在 104H 单元的值为 108 H，106 H 单元的值为 10AH。又如：

```
ARRAY   DB    12,34,56
LEN     EQU   $-ARRAY
```

那么 LEN 的值就是 ARRAY 变量所占的字节数。

2. 偶对齐伪指令 EVEN

EVEN 伪指令使当前偏移地址指针指向偶数地址，即若原地址指针已指向偶地址，则不作调整；否则将地址指针加 1，使地址指针偶数化。EVEN 可以对齐字量数据。

格式：

```
EVEN
```

EVEN 使它后面的数据或指令从偶地址开始。假设有下列变量定义，并且变量 B1 的偏移量是偶数，其内存单元分布如图 5.6 所示。

```
B1    DB   12H              ;为了表示方便，不妨再假设其偏移量为 xxxx0H
W1    DW   4567H
```

在上述定义情况下，在许多微机系统中，当需要读变量 W1 及其后面的字内容时，硬件将按图 5.7 所示的方式分两次读出该字内容，再拼接成一个字内容。这时，无疑需要两个读内存周期，从而影响程序执行的速度。

出现上述问题的主要原因就是，字变量 W1 在数据段内的偏移量是奇数，为了保证其偏移量是偶数，需要在其定义之前加上伪指令 EVEN。所以，可把前面的变量定义改变成下列形式：

```
B1    DB   12H
EVEN
```

```
W1    DW  4567H
```

这时，变量的内存分配和读取字变量 W1 的过程如图 5.8 所示。

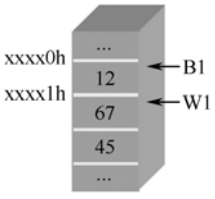


图 5.6　内存单元分布图

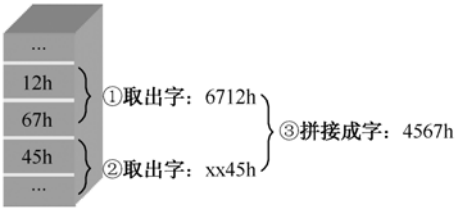


图 5.7　奇偏移量读取一个字的过程示意图

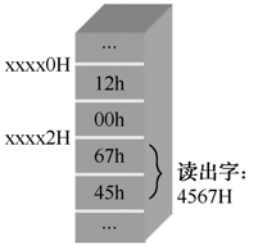


图 5.8　偶偏移量读取字过程

3. 对齐伪指令 ALIGN

ALIGN 伪指令是将当前偏移地址指针指向 n（n 是 2 的乘方）的整数倍的地址，即若原地址指针已指向 n 的整数倍地址，则不作调整；否则将指针加以 l～n−l 中的一个数，使地址指针指向下一个 n 的整数倍地址。

格式：

```
ALIGN     n
```

其中，

n 值是 2 的乘方（2，4，8，16，…），而且要小于所在段的定位属性值。

ALIGN n 伪指令的作用是告诉汇编程序，本伪指令下面的内存变量必须从下一个能被 n 整除的地址开始分配。如果下一个地址正好能被 n 整除，那么，该伪指令不起作用；否则，汇编程序将空出若干个字节，直到下一个地址能被 n 整除为止。

例如：

```
DATA    SEGMENT          ;完整段定义默认采用 PARA 节定位属性，其值为 16 的倍数
DATA1   DB      1,2,3    ;DATA1 的偏移地址为 0000H
        EVEN             ;等价于 ALIGN 2
DATA2   DW      5        ;DATA2 的偏移地址为 0004H
        ALIGN   4
DATA3   DD      6        ;DATA3 的偏移地址为 0008H
        ORG     $+10H
DATA4   DB      'abc'    ;DATA4 的偏移地址为 00lCH
DATA    ENDS
```

上例中 EVEN 等价于 ALIGN 2，ALIGN 4 可以使下一个偏移地址开始于双字边界，即对齐了双字边界，DATA3 的偏移地址为 0008H。

注意：

ORG、EVEN 和 ALIGN 指令也可在代码段使用，用于指定随后指令的偏移地址。

5.4.5　标题伪指令 TITLE

标题定义伪指令 TITLE 是给程序指定一个标题，以便在列表文件中每一页的第一行都显示这个标题。

格式：

```
TITLE  标题名
```

其中，标题名是程序设计人员任选的不加引号的字符或字符串，最长为 60 个字符。一个程序模块中只允许一个 TITLE 命令，否则会引起错误。

如果程序中没有使用 NAME 伪操作，则汇编程序将用“标题字符串”的前 6 个字符串作为其模块名。如果程序中既无 NAME 伪操作，也无伪指令 TITLE，那么，源文件名将作为模块名。

例如：

```
TITLE     MODULE1
DATA      SEGMENT
STRING    DB      'CHARACTER DISPLAY   S'
DATA      ENDS
STACKl    SEGMENT PARA   STACK
DW        20H   (0)
STACKl    ENDS
PUBLIC    STRING
EXTRN     DISPLY: FAR
CODEl     SEGMENT
          ASSUME CS: CODEl, DS: DATA, SS: STACK1
 START:   …
          CALL      DISPLY
          …
CODEl     ENDS
          END   START
```

5.4.6 程序结束伪指令 END

伪指令 END 标志源程序结束，并指定程序运行时的起始地址。它一方面告诉汇编程序，END 伪指令后面的任何语句在汇编时都被略去，另一方面又告诉汇编程序，在程序装入内存时根据起始地址（如一个标号）的段基值和偏移量分别自动装入 CS 和 IP 中。

格式：

```
END       起始地址
```

其中，起始地址可以是一个标号，也可以是一个表达式。它是程序装入内存后，开始执行的起始点。

任何一个源程序都以 END 伪指令结束，END 伪指令告诉汇编程序源程序汇编到此结束。在汇编语言程序设计中，通常在程序执行的第一条语句（不一定是指令语句）设置一个标号，然后在源程序最后用 END 伪指令结束。

例如：

```
                …
CODE            SEGMENT
ASSUME          CS:    CODE, …
START:          …
```

```
                ...
CODE            ENDS
                END   START
```

5.4.7　模块连接伪指令 PUBLIC 和 EXTRN

在开发一个功能较弱的应用程序时，其执行文件通常可以由一个目标文件（模块）连接而成，当开发一个功能较强、关系较复杂的应用程序时，其执行文件很难由一个目标文件（模块）连接而成，而常常是由多个目标文件（模块）连接而成的。各模块之间无疑会存在着相互调用、相互访问数据单元等内在联系，各模块之间的相互联系就产生了如何在源程序中来表达这种联系的问题。

为了描述各模块之间的联系，汇编语言提供了两条伪指令 PUBLIC 和 EXTRN，它们的作用有点像 C/C++语言说明变量、过程和函数是“全局的”或“外部的”。这两条伪指令的具体用法和含义如下。

1. 伪指令 PUBLIC

伪指令 PUBLIC 是用来说明当前模块中哪些标识符是能被其他模块引用的公共标识符。其说明的一般格式如下。

格式：

```
PUBLIC   标识符 1, 标识符 2, …
```

其中，

（1）PUBLIC 伪指令可以出现在本模块源程序的任何位置。

（2）标识符可以是变量名、过程名、标号、用等值“EQU”或等号“=”定义的常数等，各标识符之间要用逗号分开。

PUBLIC 伪指令说明了标识符 1、标识符 2 等是公共标识符，可以在其他程序模块中直接引用。在一个模块中，可用多条 PUBLIC 伪指令来说明公共标识符。

2. 伪指令 EXTRN

伪指令 EXTRN 是用来说明在当前模块所使用的标识符中，哪些标识符是已在其他模块中被定义为指定类型的标识符。

伪指令 EXTRN 的一般说明格式如下。

格式：

```
EXTRN   标识符 1:类型 1, 标识符 2:类型 2, …
```

其中，

（1）“标识符”和“类型”之间要用冒号“:”连接。

（2）标识符 1、标识符 2 等是外部标识符，它们在其他模块中已被分别定义为类型 1、类型 2 等，各符号后的类型是不能省略的。如果符号为变量，那么类型为 BYTE、WORD、DWORD、结构名、记录名；如果符号为标号或过程名，那么类型为 NEAR 或 FAR；如果符号为用等值“EQU”或等号“=”定义的常数，那么类型为 ABS。如果在一条说明伪指令中说明了多个标识符，那么，各标识符之间要用逗号分开。

在一个模块中，可用多条 EXTRN 伪指令来说明本模块所引用的外部标识符。如果当前模块使用了其他模块的标识符，而对它又没有加以说明的话，那么，汇编程序将会给出

下列出错信息：

```
error nnnnn: undefined symbol : XXXXXX
```

其中，“nnnnn”是错误号，“XXXXXX”是当前模块中没有定义的标识符。

注意：

伪指令 EXTRN 中所说明的标识符必须在其定义的模块中被 PUBLIC 伪指令说明为公共标识符，并且其说明的标识符类型要与该标识符定义的类型相一致；否则，要么不能生成其可执行文件，要么其执行文件不能正确运行。

PUBLIC 和 EXTRN 伪指令在模块化程序设计中是配合使用的，一个全局符号必须在定义它的程序模块中用 PUBLIC 伪指令加以说明，然后在使用它的其他程序模块中用 EXTRN 伪指令加以说明。也就是说，凡是在 EXTRN 伪指令中出现过的符号一定在某程序模块的 PUBLIC 伪指令中说明，它们两者必须是相互呼应的。

下面是两个程序模块连接的例子，模块 MODULE1 源程序如下：

```
            TITLE   MODULE1
 DATA       SEGMENT
VAR1        DB   'I am a student '
DATA        ENDS
STACKl      SEGMENT      PARA      STACK
            DW  20H   DUP (1)
STACKl      ENDS
PUBLIC      STRING
EXTRN       EXT1: FAR
CODEl       SEGMENT
            ASSUME CS: CODEl, DS: DATA, SS: STACK1
 START:     ...
            CALL      EXT1
            ...
CODEl       ENDS
            END       START
```

模块 MODULE2 源程序如下。

```
            TITLE   MODULE2
PUBLIC      EXT1
EXTRN       VAR1: BYTE
CODE2       SEGMENT
EXT1        PROC      FAR
            ...
            MOV DX, OFFSET   VAR1
            ...
            RET
EXT1        ENDP
CODE2       ENDS
            END
```

上例中 MODULE1 模块是主程序，它调用 MODULE2 中的子程序 EXT1。这两个程序模块中分别用 PUBLIC 伪指令说明变量名 VAR1 和过程名 EXT1 为全局符号，用 EXTRN

伪指令说明 VAR1 和 EXT1 为外部符号。这样，两个程序模块先分别汇编，得到两个目标文件 MODULEl.OBJ 和 MODULE2.OBJ，然后再用连接程序（LINK.EXE）把这两个目标文件连接起来，形成一个可执行文件。

5.5　汇编语言源程序正常结束方式

经过汇编和连接两个步骤，将汇编语言源程序生成一个可在 DOS 下直接执行的文件（后缀名为 EXE 文件），在 DOS 下执行这个文件时，DOS 像调用子程序一样，把控制权交给它，而这个 EXE 文件执行完成后也要像子程序返回调用程序一样结束程序执行，返回 DOS。

在.COM 文件中和 DEBUG 下所写程序，可以使用 INT 20H 或 INT 3H 来终止程序。而对于 EXE 类型的应用程序，可以使用 INT 21H 系统功能调用来终止程序，也可以使用程序段前缀来终止程序，但后者必须将执行模块定义为过程。

5.5.1　使用 4CH 系统功能调用正常结束

DOS 为程序设计人员提供了很多系统功能调用，其中调用号为 4CH 的系统调用的功能是结束程序执行返回 DOS，因此可以利用它来完成程序返回操作系统，其具体方法是在程序结束时，使用以下两条指令：

```
MOV     AH, 4CH
INT     21H
```

例如：

```
…
CODE    SEGMENT
        ASSUME  CS:CODE, …
START:   …                          ; 程序主体
        MOV     AH, 4CH             ; 4CH 号系统功能调用
        INT     21H                 ; 结束程序，返回 DOS 系统
CODE    ENDS
        END     START
```

用 4CH 系统功能调用返回 DOS 的方法只需在程序主体结束后写上两条指令，不需要编写额外的代码，实现方便，易于理解。但并不是任何程序都可以用这种方法来实现返回 DOS 的，只有返回 DOS 的主程序才能使用这种方式，而用户调用的子程序应该返回调用程序，不能用 4CH 系统功能调用实现返回。

5.5.2　使用程序段前缀正常结束

对于任一装入内存的可执行文件（EXE 文件），DOS 都将在代码段前 100H 字节中建立程序段前缀 PSP (Program Segment Prefix)，在其后装入该 EXE 程序，并把控制权交给它（即源程序结束伪指令 END 后的参数指向的远过程）。PSP 的内容主要包括三部分信息：

被装入程序与 DOS 连接时使用的信息、供装入程序使用的参数和供 DOS 本身使用的信息。

在 DOS 转移控制权时，将 CS 指向 EXE 程序的代码段，SS 指向堆栈段，DS 和 ES 并不指向用户程序的数据段和附加段，而是指向 PSP，这样便于用户使用和处理 PSP 中的信息。因此，EXE 文件装入内存后，存储器的分配情况如图 5.9 所示。

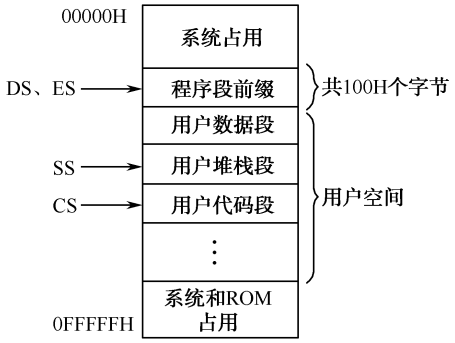


图 5.9　EXE 程序装入内存后的存储器分配情况示意图

对于 EXE 类型的应用程序而言，其代码段与 PSP 不是同一个段，所以不能简单地直接利用 INT 20H 来终止程序。DOS 在 PSP 的偏移 0 处，安排了 INT 20H 指令。应用程序只要设法转到 PSP 的偏移 0 处，就能实现程序的终止，返回 DOS。其具体的步骤如下。

（1）把用户程序编制成一个过程，并且将过程的类型属性设置为 FAR。

（2）将 PSP 的起始逻辑地址压入堆栈，即将“INT 20H”指令的地址压入堆栈。由于 EXE 文件装入内存时 DS 和 ES 都是指向 PSP 的，因此用指令“PUSH DS”就可以实现将 PSP 的起始逻辑地址（INT 20H 的逻辑地址）压入堆栈的操作。

（3）在用户程序编制而成的过程结束时使用返回指令 RET，这样执行该指令时系统便会将保存在堆栈中的 PSP 起始逻辑地址弹出到 CS 和 IP 中，进而转去执行 PSP 首地址处的指令 INT 20H，从而实现返回 DOS 的目的。

例如：

```
…
CODE    SEGMENT
PROCl   PROC    FAR                ; 使用过程，主程序开始
        ASSUME  CS:CODE,  …
START:  PUSH    DS                 ; 保存 PSP 首地址，RET 返回时将其装入 CS
        MOV     AX, 0              ; 设置偏移地址为 0
        PUSH    AX                 ; 入栈保存偏移地址
        …
        RET                        ; 用过程返回 DOS
PROC1   ENDP                       ; 主程序结束
CODE    ENDS
        END     START
```

用程序段前缀实现程序结束返回 DOS 的方法是利用软中断指令“INT 20H”实现的一种间接返回方式，实现起来稍显复杂，不容易理解，但无论是用户调用的子程序或 DOS 调用的主程序都可以用这种方式来实现返回，克服了 4CH 系统功能调用只能供返回 DOS

的主程序调用的缺点。

习题

1．简述变量和标号的含义，两者有何区别？

2．变量和标号有什么属性？

3．判断下列标识符的合法性。

（1）code　（2）ABCDH　（3）eax　（4）A?@

（5）www.bai.edu.cn　（6）_BX　（7）Data

（8）1abcH　（9）???01　（10）www@zsu　（11）AX$DX

4．按下面要求写出相应的数据定义语句（未指定变量名的，可任意指定或默认）。

（1）定义一个字节区域，第一个字节的值为 20，其后跟 20 个初值为 0 的连续字节。

（2）定义一个以 0 为结束符的字符串，其初值为：The courses are easy。

（3）定义一个以'$'为结束符的字符串，该串中含有换行符和回车符。

（4）定义 100 个字，其初值为 0。

（5）定义一个双字变量，其初始值为 7845H。

（6）定义一个十进制数字节变量，其初始值为 67。

（7）定义一个十六进制数字变量，其初始值为 FFF0。

（8）从一个偶地址开始定义一个字变量 word；

（9）先定义一个字符串 String，其偏移量指定为 10，其初值为"ABCDEF"，然后再定义 3 个字类型的数组 Data，要求 String 和 Data 的偏移量相同。

（10）定义一个字符串 String，其初值为"01234567"，再定义 4 个字类型的数组 Data，要求 String 和 Data 共享同一片内存单元。

5．画图说明下列语句所分配的存储空间及初始化的数据值。

（1）ARR1　DB　8, 67H, -3

（2）ARR2　DB　5D2CH, 2, ?

（3）ARR3　DB　2 DUP(0, 1, 2 DUP(3, 4))

（4）BYTEVAR　DB　‘BYTE’, 12, 32H, 5 DUP(0, ?, 3 DUP(1, 3), ?)

（5）WORDVAR　DW 3 DUP(0, 0, 1), 3, 5, ‘WO‘, ‘RD‘, 345H

6．伪指令和汇编指令的本质区别是什么？伪指令有什么主要用途？

7．伪指令 ASSUME 的作用是什么？

8．伪指令 EQU 与“=”伪指令的区别是什么？

9．怎样规定一个程序执行的开始位置？主程序执行结束后应该如何返回 DOS 系统？

10．设数据段数据定义如下：

```
DATA      SEGMENT
NUM1      DW        2010H
NUM2      DW        15 DUP(2010H)
STRING1   DB        ‘STUDENT’
BUF       DB        10H DUP(1,2 DUP(0))
DATA      ENDS
```

下面几条 MOV 指令单独执行后，寄存器 BX 中的内容是什么？

（1）MOV　　BX,NUM1

（2）MOV　　BX,NUM2

（3）MOV　　BX,TYPE　　NUM1

（4）MOV　　BX,TYPE STRING1

（5）MOV　　BX,LENGTH NUM2

（6）MOV　　BX,LENGTH STRING1

（7）MOV　　BX,SIZE NUM2

（8）MOV　　BX,SIZE BUF

11．若某程序数据段中有数据定义如下：

```
DATA1    EQU    23H
DATA2    DW     23H
```

则下面两条指令有何异同？

```
MOV AX,DATA1
MOV BX,DATA2
```

12．若某程序数据段中有数据定义如下：

```
DATA1    DW     ?
DATA2    DB     16 DUP(?)
COUNT    DD     ?
LEN1     EQU    COUNT-DATA2
LEN2     EQU    $-DATA1
```

经过汇编后，LEN1 和 LEN2 的值是多少？各表示什么意义？

第 6 章

8086/8088 常用指令

从本章开始将详细介绍各种指令的格式、功能和用法。指令是计算机用以控制各部件协调动作的命令，指令系统是指微处理器所能执行的各种指令集合，微处理器的主要功能是通过它的指令系统来实现的。不同的微处理器拥有不同的指令系统，在其他条件相同的情况下，指令系统越强，计算机的功能也就越强。

Intel 8086/8088 CPU 指令系统丰富，功能强大，大多数指令既能处理字数据，也能处理字节数据。

指令按照书写格式，可分为以下三大类：

（1）双操作数指令：OPR DEST,SRC；

（2）单操作数指令：OPR DEST；

（3）无操作数指令：OPR；

其中，OPR 为指令操作码，DEST 为目标操作数，SRC 为源操作数。

指令按功能划分可以分为以下几类：

（1）数据传送类指令（Transfer Instructions）；

（2）算术运算类指令（Arithmetic Instructions）；

（3）位操作类指令（Boolean Instructions）；

（4）字符串处理类指令（String Instructions）；

（5）控制转移类指令（Control Transfer Instructions）；

（6）处理器控制类指令（Processor Control Instructions）。

本章按照功能介绍其中的 4 类指令，控制转移类指令在第 7 章节介绍，字符串处理类指令在第 10 章中介绍。

6.1 数据传送类指令

传送类指令负责在寄存器、存储单元或 I/O 端口之间传送数据和地址，是最简单、最常用的一类指令。可以分为 7 类数据传送指令：通用数据传送指令、堆栈操作指令、数据交换指令、查表转换指令、地址传送指令、标志位传送指令和端口输入/输出指令。

6.1.1 通用数据传送指令

通用数据传送指令（MOV）是使用得最频繁的指令，它的功能是把一个字节或一个字从源操作数传送至目的操作数，当 MOV 指令执行后，源操作数和目的操作数将具有相同的内容，源操作数保持不变，MOV 指令对标志寄存器各位都没有影响，其格式如下：

格式：

```
MOV     DEST，SRC
```

其中，

（1）SRC 为源操作数，可以是累加器、寄存器、存储单元和立即数。

（2）DEST 为目的操作数，可以是累加器、寄存器和存储单元。

通用数据传送指令不改变源操作数。通用数据传送指令可实现的传送方向如图 6.1 所示，其中虚线表示通用寄存器传送至段寄存器时，CS 不能作目的操作数。

对 MOV 指令有以下几条具体规定，其中有些规定对其他指令也同样有效。

（1）两个操作数的数据类型要相同，要同为 8 位或 16 位，例如，MOV　BL，AX 是不正确的；

（2）两个操作数不能同时为段寄存器，例如，MOV　ES, DS 是错误的；

（3）代码段寄存器 CS 不能为目的操作数，但可作为源操作数，例如，指令 MOV　CS, AX 不正确，但指令 MOV　AX, CS 是正确的；

（4）立即数不能直接传给段寄存器，例如，MOV　DS, 100H 是错误的；

（5）立即数不能作为目的操作数，例如，MOV　100H, AX 是错误的；

（6）指令指针 IP 不能作为 MOV 指令的操作数；

（7）两个操作数不能同时为存储单元，例如，MOV　VAR1, VAR2 是错误的，其中 VAR1 和 VAR2 是同数据类型的内存变量。

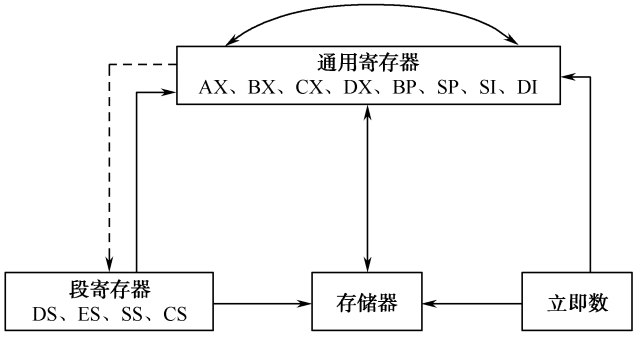


图 6.1　MOV 指令数据传送方向示意图

具体地说，MOV 指令能实现下列传送功能。

1. CPU 内部寄存器之间的数据传送

例如：

```
MOV     AH,AL          ;字节传送，(AL)→AH
MOV     BX,CX          ;字传送，(CX)→BX
MOV     BP,SP          ;字传送，(SP)→BP
```

```
MOV     AX,CS                   ;字传送，(CS)→AX
```

2. 立即数送至通用寄存器或存储单元（各种存储器寻址方式）

立即数传送到通用寄存器是指将立即数传送给 AX、BX、CX、DX、BP、SP、SI、DI，以及 AH、AL、BH、BL、CH、CL、DH、DL 寄存器，但不能传送到段寄存器。

例如：

```
MOV     AL,2
MOV     SI,-4
MOV     VARW,1234H              ;VARW 是字变量
MOV     WORD PTR[SI],3482H
MOV     DL, 'A'                 ; 65→DL
MOV     VAR[BX] [DI+3] ,20H     ;20H 同 VAR 变量的类型一致
```

注意：

立即数和寄存器数据类型必须一致，立即数可以是二进制、八进制、十进制、十六进制常数，也可以是带符号或无符号的整数，当然也可以是 ASCII 码字符。

3. 寄存器与存储器之间的数据传送

寄存器与存储器之间的数据传送是指除了 CS 和 IP 以外的所有寄存器与内存储器之间的数据传送。

例如：

```
MOV     AL, VAR             ;(VAR)→AL
MOV     BH, [DI]            ;存储器操作数为寄存器间接寻址
MOV     [DI], DX            ;(DX)→(DS) ×l0H+ (DI)
MOV     [BX][DI],DL         ;(DL)→(DS) ×10 H+(BX)+(DI)
MOV     VARB,BL             ;VARB 是一个字节变量
MOV     SI, ES: [BP]        ;使用段超越前缀，((ES) ×l0H+ (BP))→SI
MOV     AX, 10H[BX+SI]      ;((DS) ×10H+(BX)+( SI)+10H)→AX
```

上例中可以看出，对于存储器操作数，可采用各种存储器寻址方式。关于 MOV 指令，除了上面所说的遵守规定外，还要注意的是，源操作数和目的操作数不能同时是存储器操作数（除串操作指令外）。如果要在两个存储单元传送数据，可以利用通用寄存器过渡的方法进行。

例如：

```
MOV     AX,VARW1            ;VARW1 为字变量
MOV     VARW2,AX
```

当然这种利用通用寄存器过渡的方法也可以用于段寄存器间的数据传送。

例如：

```
MOV     AX,CS
MOV     DS,AX
```

【例 6.1】　判断如下指令是否正确。

```
MOV     [1000H],[3000H]
MOV     DS,300
MOV     AX,BL
MOV     ES,DS
```

这四条指令都是错误的，原因是违反了传送指令的规则。

第 1 条指令：错在两个操作数不能同时为存储器操作数；

第 2 条指令：错在立即数不能传送到段寄存器；

第 3 条指令：错在类型不匹配；

第 4 条指令：错在段寄存器不能直接传送段寄存器。

说明：

（1）MOV [1000H],[3000H]可以修改为：

```
MOV     AX,[3000H]
MOV     [1000H],AX
```

（2）MOV DS,300 可以修改为：

```
MOV     AX,300
MOV     DS,AX
```

（3）MOV ES,DS 可以修改为：

```
MOV     AX,DS
MOV     ES,AX
```

6.1.2 堆栈操作指令

在 8086/8088 系统中，堆栈是指在存储器中开辟的一个特殊存储区，称为“堆栈段”，如图 6.2 所示。堆栈主要用来临时存放数据备用，也常在执行调用子程序（CALL）或中断指令时，用堆栈来保存返回地址，所以堆栈在程序运行时是必不可少的。栈顶的一端地址较小，栈底的一端地址较大。堆栈的段值在堆栈段寄存器 SS 中，堆栈指针寄存器 SP 始终指向栈顶，只要重新设置 SS 和 SP 的初值，就可以改变堆栈的位置，堆栈的深度由 SP 的初值决定。堆栈有进栈（存入数据）和出栈（取出数据）两种操作，编程时常成对使用。

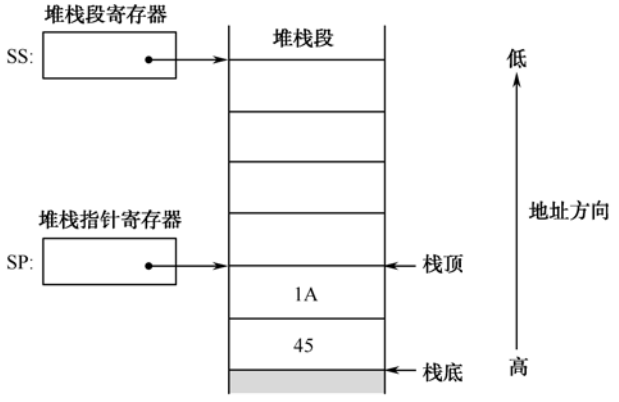


图 6.2　堆栈示意图

1. 进栈指令 PUSH

进栈指令 PUSH 用于把源操作数压入堆栈，源操作数只能是字，其格式如下：

格式：

```
PUSH    SRC
```

其中，SRC 为源操作数，且源操作数只能为字类型。

该指令把源操作数 SRC 压入堆栈，首先将栈顶指针值减 2，即(SP)-2→SP，然后将源操作数送入 SS:SP 所指的栈顶。源操作数可以是字通用寄存器、段寄存器和字存储器。

例如：

```
PUSH    AX              ; (SP)-2→SP，(SP)←(AL)，(SP+1)←(AH)
PUSH    SI              ; SP←SP-2，(SP)，(SP+1)←(SI)
PUSH    SS              ;将(SS)压入堆栈
PUSH    [SI]            ;SP←SP-2，(SP)←([SI])，(SP+1)←([SI]+1)
PUSH    DATA[BX] [SI]   ;将 DATA[BX+SI]一个字的内容压入堆栈
```

注意：

数据进入堆栈时遵守“高高低低”原则，即高位数据放在高字节中，低位数据放在低字节中。

2. 出栈指令 POP

出栈指令 POP 用于从栈顶弹出一个字数据到目的操作数，SP 随着出栈而增大，其格式如下：

格式：

```
POP     DEST
```

其中，DEST 为目的操作数。

该指令从栈顶弹出一个字数据到目的操作数，首先将堆栈顶由 SS:SP 所指向的字送入目的操作数，然后将堆栈指针 SP 加 2，即(SP)+2→SP。每次操作完，SP 总是指向栈顶。目的操作数可以是通用寄存器、段寄存器（除 CS 外）和字存储单元。

例如：

```
POP     [SI]            ; (SP+1)，(SP)→(SI)，SP←SP+2
POP     DATA[BX]
POP     SS
POP     AX
```

【例 6.2】 假设 AX=1234H，分别执行“PUSH AX”和“POP BX”指令操作过程是怎样的？

堆栈操作指令 PUSH 和 POP 的操作示意图如图 6.3 所示。

因为 AX=1234H，执行指令“PUSH AX”，进栈前后堆栈段的情况如图 6.3(a)、(b)所示，执行指令“POP BX”，出栈前后堆栈段情况如图 6.3(b)、(c)所示。其执行步骤可简单表示为：

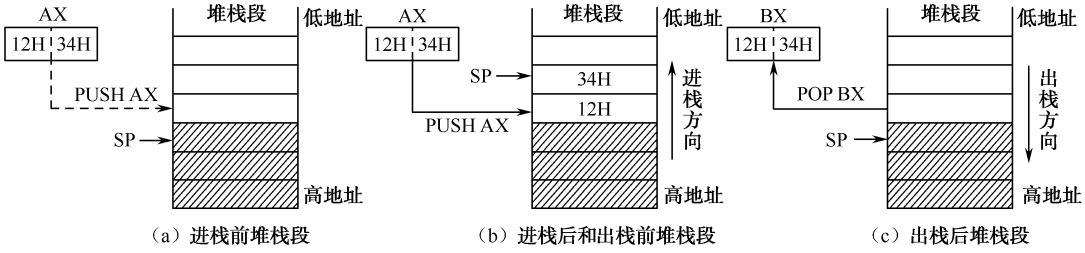


图 6.3　进栈和出栈示意图

（1）(SP)−2→SP，(SP)←(AL)，(SP+1)←(AH)

（2）(SP+1)，(SP)→BX，SP←SP+2

注意：

（1）SP 总是指向栈顶，而用 PUSH 和 POP 指令存取数时都是在栈顶进行的，所以堆栈是“先进后出”或称为“后进先出”。因为栈底在高地址，堆栈是从高地址向低地址延伸的，所以栈底就是最初的栈顶。

（2）PUSH 指令是先修改 SP，使 SP 指向刚刚释放的空闲单元，然后将内容压栈。而 POP 指令是先从栈顶弹出一个字，然后将堆栈指针(SP)+2→SP，以便将刚刚弹出的堆栈字空间释放。

（3）PUSH CS 是合法的，而 POP CS 是非法的。

（4）PUSH 和 POP 指令的操作数都不能是立即数。

（5）PUSH 和 POP 指令都不影响标志位。

6.1.3 数据交换指令

数据交换指令 XCHG 可以方便地实现通用寄存器与通用寄存器或存储单元之间的数据交换。其格式如下。

格式：

```
XCHG    OPRD1，OPRD2
```

其中，OPRD1 和 OPRD2 为操作数，必须一个是通用寄存器（不包括段寄存器），另一个是通用寄存器（不包括段寄存器）或存储单元，但不能同时为存储单元，也不能包含立即数。两个操作数必须同时为字节或字。

例如：

```
XCHG    AX, BX          ; AX 与 BX 寄存器内容交换
XCHG    SI,BX           ; SI 与 BX 内容交换
XCHG    AL, BL          ; AL 与 BL 内容交换
XCHG    VARW,CX         ; 变量 VARW 与 CX 寄存器内容交换
XCHG    DH,VARB         ; DH 与变量 VARB 内容交换
XCHG    [SI+3],AL       ; DS: [SI+3]所指的内容与 AL 内容交换
XCHG    [DI+BP+3],BX
```

【例 6.3】 假设 DS=2000H，SI=2479H，AL=86H，DI=1716H，BX=4232H，(2247CH)=55H。问执行下列指令后的结果？

```
XCHG    [SI+3],AL       ; 存储器与寄存器之间交换数据
XCHG    DI,BX           ; 寄存器之间交换数据
```

存储单元[SI+3]的物理地址为 PA=20000H+2479H+3H=2247CH，而(2247CH)=55H，所以执行完第一条指令后，存储器与寄存器之间交换数据，AL=55H，(2247CH)=86H。执行完第二条指令后，寄存器之间交换数据，DI=4232H，BX=1716H。

6.1.4　查表转换指令

查表转换指令 XLAT 常用来实现单列表格的查表操作，指令有两个隐含操作数 BX 和 AL。其格式如下所示。

格式：

```
XLAT    或    XLAT  OPRD
```

其中，OPRD 为操作数，长度不得超过 256 个字节。

查表指令 XLAT 常用来实现单列表格的查表操作，首先把单列表格存放在存储器中，即将表格的开头地址存放在基址寄存器 BX（16 位）中，待查元素在表格中的位置存放在 AL（8 位）中，然后执行本指令，结果是将表格中查到元素的内容送入 AL 中。

该指令的操作数是隐含的，目的操作数是 AL，源操作数是用变址方式寻址的存储单元，即 BX 和 AL 的内容之和（DS:[BX+AL]→AL）指定的存储单元。XLAT 指令执行过程示意图如图 6.4 所示。

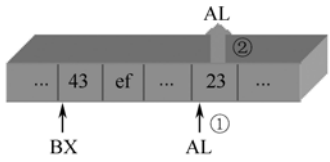


图 6.4　XLAT 指令执行过程示意图

【例 6.4】 利用 XLAT 指令编写查询小写字母的 ASCII 码一个小程序段。

小程序段如下：

```
DATA    SEGMENT                              ;定义数据段
TABLE   DB  'abcdefghijklmnopqrstuvwxyz' ;定义表格变量（字节字符变量，8 位）的首地址
…
DATA    ENDS
…
MOV     BX,OFFSET TABLE
MOV     AL,05H
XLAT
…
```

说明：

（1）数据段中定义的表格 TABLE 为字节变量，MOV BX,OFFSET TABLE 指令是将变量 TABLE 的偏移地址（表格的开头地址）送入 BX 中。

（2）MOV AL,05H 指令是将待查元素在 TABLE 表中的位置（第 6 项）05H 送入 AL 中。

（3）XLAT 指令是将查表后，f 的 ASCII 码 66H 送入 AL 中，该指令也可以写为 XLAT TABLE。

注意：

BX 寄存器隐含在数据段 DS 中，如果 TABLE 定义在其他段（如在扩展段 ES），那么查表指令改为 XLAT　ES: TABLE。

6.1.5　地址传送指令

地址传送指令有“LEA”、“LDS”和“LES”3 种。

1. LEA 指令（Load Effective Address）

LEA 指令称为有效地址传送指令，把源操作数的有效地址传送至目的操作数寄存器，其格式如下。

格式：

```
LEA     REG,OPRD
```

其中，REG 为 16 位的通用寄存器（即 AX、BX、CX、DX、SP、BP、SI、DI 之一），OPRD 源操作数是存储器操作数，如变量、标号或地址表达式。

例如：

```
LEA     AX,BUFFER               ; 将变量 BUFFER 的符号地址（偏移地址）送入寄存器 AX
LEA     CX,BUFARRAY[BX][SI]     ; 将数组元素的位移量送入 CX 中
LEA     DX,[BX+3]               ; 将操作数的偏移地址送入寄存器 DX
LEA     AX,[BP][DI]             ; 将有效地址(BP)+(DI)送入 AX 中
```

【例 6.5】 假定有 DS=1000H，VAR=300H，(10300H)=3278H，其中符号地址 VAR 可以是变量或标号，执行以下两条指令的结果如何？

```
LEA     AX,VAR
MOV     AX,VAR
```

指令 LEA AX,VAR 是将 VAR 单元的偏移地址传送到 AX，于是，执行后 AX=300H；指令 MOV BX,VAR 是将 VAR 所指单元的内容传送到 AX，于是，执行后 AX=3278H。

说明：

指令 MOV BX,OFFSET VABLE 和 LEA BX,VABLE 是等价的，都是将源操作数的有效地址传送至目的操作数寄存器。

2. LDS 指令（Load pointer into DS）

LDS 指令是用于把源操作数中包含的 32 位地址指针段首址部分（高位字）送到数据段寄存器 DS，把偏移地址（低位字）送到通用寄存器中。其格式如下：

格式：

```
LDS     REG,OPRD
```

其中，

（1）REG 为目的操作数，表示 16 位的通用寄存器之一。

（2）OPRD 为源操作数，表示双字的各种寻址方式的存储器操作数的首地址。

例如：

```
LDS     AX,VARDD            ; WARDD 为双字变量
LDS     DI,[BX]
```

假如双字变量 VARDD 包含的 32 位地址指针的段值为 1000H，位移量为 12H，则执行指令“LDS BX, VARDD”后，数据段寄存器 DS 的值为 1000H，寄存器 AX 的值为 12H。位移量存储在双字变量的低地址字中，段值部分存储在高地址字中。

假设(DS)= 4000H，(BX)=1234H，(41234H)=88H，(41235H)=56H，(41236H)=BCH，(41237H)= 9AH，则(DI)= 5688H，(DS)= 9ABCH。

3. LES 指令（Load pointer into ES）

LES 指令用于把源操作数中包含的 32 位（4 字节）地址指针的段首址部分（高位字）送到附加段寄存器 ES，把偏移部分（低位字）送到通用寄存器中。其格式如下：

格式：

```
LES     REG，OPRD
```

其中，

（1）REG 为目的操作数，表示 16 位的通用寄存器之一。

（2）OPRD 为源操作数，表示双字的各种寻址方式的存储器操作数的首地址。

LES 指令除了把双字指针中的段基址传送至 ES 段寄存器，而不是传送至 DS 外，其他与 LDS 指令完全相同。

例如：

```
LDS     AX,VARDD          ; WARDD 为双字变量
LDS     DI,[BX]
```

假如双字变量 VARDD 包含的 32 位地址指针的段值为 1000H，位移量为 12H，则执行指令“LDS BX, VARDD”后，附加段寄存器 ES 的值为 1000H，寄存器 AX 的值为 12H。位移量存储在双字变量的低地址字中，段值部分存储在高地址字中。

假设(DS)= 4000H，(BX)=1234H，(41234H)=88H，(41235H)=56H，(41236H)=BCH，(41237H)= 9AH，则(DI)= 5688H，(ES)= 9ABCH。

6.1.6　标志位传送指令

在 8086/8088 CPU 中，有“LAHF”、“SAHF”、“PUSHF”和“POPF”4 种专用于标志寄存器传送指令。

1. LAHF 指令（Load AH with Flags）

LAHF 指令将标志寄存器的状态标志位 SF、ZF、AF、PF、CF 传送至 AH 寄存器的对应位，AH 寄存器的其余 3 位内容不变，LAHF 采用固定的寻找方式，其格式如下。

格式：

```
LAHF
```

LAHF 指令本身不影响这些标志和其他标志，它把标志寄存器的低 8 位（包括符号标志 SF、零标志 ZF、奇偶标志 PF 和进位标志 CF）传送到寄存器 AH 的指定位，即相应地传送到寄存器 AH 的 7、6、4、2 和 0 位，其余 3 位内容不变，如图 6.5 所示。

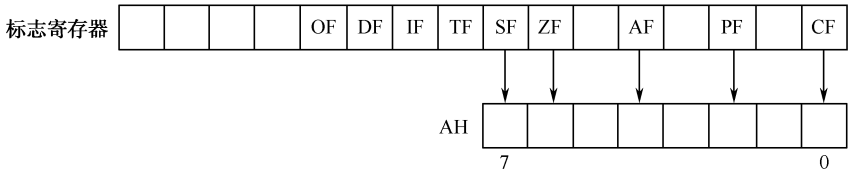


图 6.5　LAHF 指令示意图

2. SAHF 指令（Store AH into Flags）

SAHF 指令将 AH 的指定位（与状态标志对应的位）传送至标志寄存器，与 LAHF 指令传送方向相反。

格式：

```
SAHF
```

该条指令把寄存器 AH 的指定位送至标志寄存器低 8 位的 SF、ZF、AF、PF 和 CF 标志位，因而这些标志的内容会受到影响。但这条指令不影响溢出标志 OF、方向标志 DF、中断允许标志 IF 和追踪标志 TF，即不影响标志寄存器的高位字节。其操作示意如图 6.6 所示。

例如：

```
MOV     AH,0C1H
SAHF
```

执行上面两条指令后标志寄存器中 CF=1，PF=0，AF=0，ZF=1，SF=1。

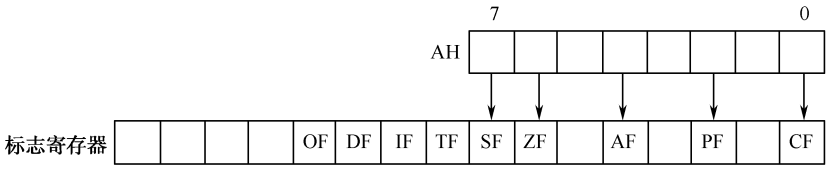


图 6.6　SAHF 指令示意图

3. PUSHF 指令

PUSHF 指令将标志寄存器的内容压入堆栈。先将堆栈指针减 2，即(SP)−2→SP，然后将整个标志寄存器（16 位）的内容压入 SP 所指向的栈顶。

格式：

```
PUSHF
```

这条指令不影响状态标志。

4. POPF 指令

POPF 指令是将标志寄存器出栈。先将堆栈顶的内容，即（(SP)+1，(SP)）传送至标志寄存器（16 位）中，然后将堆栈指针 SP 加 2，即(SP)+2→SP。

格式：

```
POPF
```

执行 POPF 指令后，标志寄存器的各位会发生相应的变化，因此，该指令影响标志寄存器。POPF 指令和 PUSHF 指令一起可以保存和恢复标志寄存器内容，即保存和恢复各标志的值。另外，这两条指令也可以用来改变追踪标志 TF。在 8086/8088 指令系统中，没有专门设置和清除 TF 标志的指令，但可以先使用 PUSHF 指令将标志压入堆栈，然后设法改变栈顶字单元中的第 8 位（把整个标志寄存器看成是一个字），再用 POPF 指令把该字弹回到标志寄存器中，这样即改变变了标志 TF。

6.1.7　端口输入/输出指令

端口输入/输出指令（I/O 指令）是 CPU 与外部设备进行通信的最基本途径，外部设备实际上是一组外部设备寄存器(包括数据寄存器、设备状态寄存器和设备控制寄存器 3 类)，I/O 端口空间只能用输入/输出(I/O)指令访问。运行有 I/O 指令的程序时，如果没有硬件端口的支持的话，计算机将有可能死机。

8086 有两条专门的 I/O 指令与端口进行通信，既可以传送字节也可以传送字，由外部

端口宽度决定，每条指令都有直接和间接两种端口寻址方式。

1. 端口输入指令 IN

当端口地址为 0～255 范围内的数时，可以使用直接寻址，从指定的外设端口读数据到 AL 寄存器，其格式如下：

格式：

```
IN   AL,PORT
     或
IN   AX,PORT
```

其中，PORT 为端口地址，限制在 0～255。

例如：

```
IN   AL,20H
```

上例中从 20H 端口输入一个字节数据送往 AL。

当端口地址可以在 256～65535 时，必须使用间接寻址，间接寻址的寄存器一定是 DX，从 8 位端口读入一个字节到 AL 寄存器中，或从 16 位端口读入一个字到 AX 寄存器中。端口地址可以是一个字节，也可以是一个字。其格式如下。

格式：

```
MOV    DX,PORT          ; 先将端口地址送入 DX 中
IN     AL,DX            ; 再将 DX 内容的低 8 位送入 AL 寄存器中
   或
MOV    DX,PORT          ; 先将端口地址送入 DX 中
IN     AX,DX            ; 再将 DX 内容送入 AX 寄存器中
```

其中，PORT 为端口地址，限制在 0～65535。

例如：

```
MOV    DX,2EDH
IN     AX,DX
```

上例中使用 DX 间接寻址从 2EDH 输入一个字节送往 AL，从 2EEH 输入一个字节送往 AH。

2. 端口输出指令 OUT

当端口地址为 0～255 范围内的数时，使用直接寻址，端口输出指令 OUT 是将 AL 或 AX 寄存器的内容传送到指定的外设端口。

格式：

```
OUT    PORT,AL
   或
OUT    PORT,AX
```

其中，PORT 为端口地址，限制在 0～255。

当端口地址为 0～65535 范围内的数时，使用间接寻址，将 AL 中的一个字节写到一个 8 位端口，或把 AX 中的一个字写到一个 16 位端口。端口输出指令 OUT 是以 DX 的内容作为端口地址，寄存器一定是 DX，将 AL 或 AX 寄存器的内容传送到该端口。

格式：

```
MOV    DX,AL            ; 先将 AL 内容送入 DX 中
OUT    DX,PORT          ; 再将 DX 内容的低 8 位送入端口地址中
```

```
        或
        MOV     DX,AX                    ; 先将 AX 内容送入 DX 中
        OUT     DX,PORT                  ; 再将 DX 内容送入端口地址中
```

其中，PORT 为端口地址，限制在 0～65535。

【例 6.6】 将数据 88H 送到 3CCH 端口中。

指令如下：

```
        MOV     DX,3CCH
        MOV     AL,88H
        OUT     DX,AL
```

注意：

（1）IN 指令的目的操作数和 OUT 指令的源操作数只能使用寄存器 AL 或 AX，若要将输入/输出端口（I/O 端口）中的内容输入到其他寄存器或者内存单元中，只能首先输入到这两个寄存器中，然后再传送。

（2）输入/输出指令中的端口号或者寄存器 DX 的内容均为端口的地址，而传送的数据则是端口中内容。

6.2 算术运算类指令

算术运算类指令是反映 CPU 计算能力的一组指令，也是编程时经常使用的一组指令。它包括加、减、乘、除及其相关的辅助指令。

该组指令的操作数可以是 8 位或 16 位。当存储单元是该类指令的操作数时，该操作数的寻址方式可以是任意一种存储单元寻址方式。

8086 提供了字节（8 位）或字（16 位）的无符号数和带符号数的加、减、乘、除 4 种基本算术运算指令。对于带符号数，用补码表示。

8086 还有一些校正操作指令，用于 BCD 码或 ASCII 码表示的十进制数的算术运算。在应用算术运算指令时，要注意其结果对标志寄存器的影响。如何识别无符号数和带符号数的加减法结果的“溢出”呢？无符号数用标志寄存器的 CF =1 来识别；带符号数用标志寄存器的 OF =1 来识别。

6.2.1 加法指令

1. 不带进位的加法指令 ADD

不带进位的加法指令 ADD 是将目的操作数与源操作数相加，结果存入目的操作数所在的寄存器或存储单元中，源操作数内容不变，并会影响 OF、SF、ZF 和 AF、PF、CF 标志位。其格式如下。

格式：

```
        ADD     DEST,SRC
```

其中，

（1）DEST 为目的操作数，SCR 为源操作数。

（2）任一操作数都不能是段寄存器；两个操作数的类型要匹配；不允许两个操作数都是立即数或都是存储器操作数。

例如：

```
ADD     DS,BX          ; 错在“段寄存器不能作为操作数”
ADD     [SI],[DI]      ; 错在“不允许两个操作数都是存储器操作数”
ADD     23H,34H        ; 错在“不允许两个操作数都是立即数”
ADD     AX,BL          ; 错在“两个操作数的类型要匹配”
```

以上四条指令都是错误的，错误原因请见指令后的注释。

【例 6.7】　ADD 指令示例。

```
ADD     AL,7           ; 将 7 与 AL 的内容相加，结果放入 AL 中
ADD     AL,AH          ; 将 AH 与 AL 的内容相加，结果放入 AL 中
ADD     [BX+SI],AX     ; (DS: [BX+SI])+(AX)→DS: [BX+SI]
ADD     AX, [BX+SI]    ; (DS: [BX+SI])+(AX)→AX
ADD     BX,VARB        ; VARB 为字节变量
ADD     VARW,SI        ; VARW 为字变量
```

说明：

不带进位的加法指令 ADD 对标志位的影响如下。

（1）当两数是无符号数时，最高位有进位，标志位 CF =1，表示其和已超过 8 位（溢出）。

（2）当两数是有符号数时，最高位是符号位，其余 7 位是数值位。若最高位和次高位都有进位，“和”的最高位仍是 1，说明“和”是负数，结果正确，超出范围的“1”应舍去，表示其“和”既没有溢出，也没有进位，所以记标志位 OF=0，CF=0；否则，最高位有进位 1，但次高位无进位，超出范围的“1”，应理解为其“和”有溢出，不是进位，故记标志位 OF=1，CF=0。两个负数的和应该是负数，最高位本应是 1，再加上符号位的“1”，才变成“10”，需要计算机进行纠错处理，所以记 SF=1，结果应该是负数，值为 0111'1111 逐位变反再加 1，即变为补码。

（3）当结果不为 0，记 ZF=0。

（4）当 1 的个数为奇数（溢出的 1 不计算在内），记 PF=0。

（5）当低 4 位无进位，记 AF=0。在补码运算时，两个同号数相加才可能产生溢出。运算结果与加数的符号不同，则为溢出。

2. 带进位的加法指令 ADC

带进位的加法指令 ADC 主要用于多字节（或多字）加法运算中，与 ADD 配合实现多倍精度数的加法。将目的操作数内容加上源操作数内容再加上 CF 进位标志，并将结果送入目的操作数中。

格式：

```
ADC     DSET, SRC
```

其中，DSET 为目的操作数，SRC 为源操作数

尽管在 8086/8088 中可以进行 16 位运算，但 16 位二进制数能表达的整数的范围还是有限的，为了扩大数的范围，仍然需要多字节运算。为了解决这个问题，一种有效的方法就是采用两个字（32 位）来存放一个数，称为“双精度数”。

当然，还可以根据需要用更多的字或字节来存放一个数，这就是“多倍精度数”。带进位加法指令 ADC 为多倍精度数的加运算提供了方便。ADC 指令会根据运算结果设置标

志寄存器中的 CF、PF、AF、ZF、SF 和 OF 位。

ADC 指令与 ADD 指令基本相同，唯一的差异是，若 ADC 指令执行前 CF=1，则两操作数相加后再加 1，若 ADC 指令执行前 CF=0，则 ADC 指令与 ADD 指令执行情况和执行结果完全相同。

带符号的多倍精度数在计算机内仍采用补码表示，其最高字节中的最高位为符号位，这样一来，就可将其余各低位字节（或字）中的数看做无符号数。因此，当两个多倍精度数相加时，其低位部分均为无符号数相加（采用 ADD 指令），只要将产生的进位加到高位字节（或字）即可。其运算的示意图如图 6.7 所示。

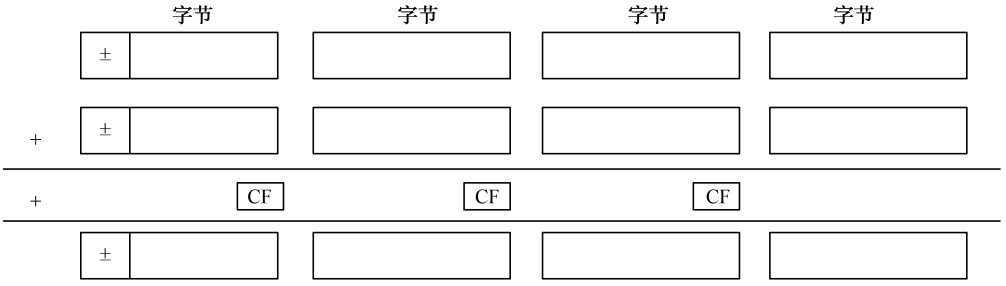


图 6.7　ADC 指令示意图

【例 6.8】　现有两个 32 位无符号数 12345678H 和 8765ABCD 相加，其和仍是一个 32 位无符号数，要求写出代码，使和的高 16 位送入 DX 中，和的低 16 位送入 AX 中。

代码如下：

```
MOV     DX,1234H
MOV     AX,5678H
ADD     AX,0ABCDH
ADC     DX,8765H
```

说明：

（1）指令“ADD　AX,0ABCDH”执行时是将两数的低 16 位相加，进位情况记录在 CF 标志位中。

（2）指令“ADC　DX,8765H”执行时是将两数的高 16 位相加，同时加上 CF 标志位，及低 16 位的进位。

（3）运算结果 999A 0245H 的高 16 位存储在寄存器 DX 中，即(DX)=999AH，低 16 位存储在寄存器 AX 中，即(AX)= 0245 H。这是两个 32 位的带符号数相加，运算结果的高 16 位习惯存储在寄存器 DX 中，低 16 位习惯存储在寄存器 AX 中。

3. 加 1 指令 INC

加 1 指令 INC 是将目的操作数的内容自加 1 后，结果送至目的操作数，格式如下。

格式：

```
INC   DEST
```

其中，

（1）DEST 为目的操作数，只能是寄存器或存储器操作数，但不能是立即数。

（2）INC 指令是一个单操作数指令，即操作对象只有一个。

INC 指令的目的操作数可以为字节或字，并被当做一个无符号二进制数，常用在循环程序中，做循环次数计数器和对地址指针进行调整，它不会影响 CF 标志位，但影响状态标志 AF、OF、DF、SF、ZF。

例如：

```
INC     AX                      ; 字数据加 1
INC     BL                      ; 字节数据加 1
INC     SI                      ; 字节数据加 1
INC     VARB                    ; VARB 是字节变量
INC     WORD  PTR [BX+SI]       ; 数据段中位移量为[BX+SI]处的字加 1
```

【例 6.9】　假设(AX)= 023FFH，则执行如下指令后的结果是什么？

```
INC  AX
```

结果：

(AX)=(AX)+1=02400H

6.2.2　减法指令

1. 不带借位的减法指令 SUB

不带借位的减法指令 SUB 将目的操作数的内容减源操作数的内容，结果存于目的操作数中，而源操作数内容保持不变，其格式如下。

格式：

```
SUB     DEST,SRC
```

其中，

（1）DEST 为源操作数，SRC 为目的操作数。

（2）任一操作数都不能是段寄存器；两个操作数的类型要匹配，即同为字数据或同为字节数据；DEST 可以是通用寄存器、存储单元或立即数，但目的操作数只能是通用寄存器或存储单元，不能是立即数，并且 DEST 和 SRC 不能同时为存储器操作数。

SUB 指令可以进行字数据的减法，也可以进行字节数据的减法。该指令影响状态标志 AF、CF、OF、PF、SF 和 ZF。

例如：

```
SUB     AH,22                   ; 字节减法
SUB     AX,BX                   ; 字减法
SUB     AL,[BX]                 ; 字节减法
SUB     VARW[BX],1234H          ; VARW 为字变量
SUB     20H[BX][SI],BL          ; 字节减法
```

【例 6.10】　执行如下指令后的结果是什么？

```
MOV     AX,5D2FH
SUB     AH,AL
```

第一条指令执行后(AX)=5D2FH，(AH)=5DH，(AL)=2FH，第二条指令执行后(AH)=2EH，执行情况如下：

$$\begin{array}{r} (AH) = 5DH = 01011101B \\ -(AL) = 2FH = 00101111B \\ \hline 00101111B \rightarrow AH \end{array}$$

说明：

（1）运算结果没有借位，所以 CF 标志位为 0。

（2）运算结果符号位为 0，为正数，所以 SF 标志位为 0。

（3）运算结果第 3 位到第 4 位无借位，所以 AF 标志位为 0。

（4）运算结果非 0，所以 ZF 标志位为 0。

（5）运算结果没有溢出，所以 OF 标志位为 0。

（6）运算结果中有 3 个“1”，所以 PF 标志位为 0

2. 带借位减法指令 SBB

带借位减法指令 SBB 是将目的操作数的内容减源操作数的内容，再减 CF 标志位，结果存于目的操作数中，其格式如下。

格式：

```
SBB  DEST,SRC
```

其中，

（1）DEST 为源操作数，SRC 为目的操作数。

（2）DEST 和 SRC 必须同时为无符号数或同时为带符号数，并且同时为字节或字。

SUB 指令与 SBB 指令基本相同，唯一的差异是，若 SBB 指令执行前 CF 标志位为 1，则两个操作数相减后再减去 1；若 CF 标志位为 0，则 SBB 指令与 SUB 指令执行情况完全相同。SUB 与 SBB 既适用于无符号二进制数的减法运算，也适用于带符号二进制数的减法运算。SBB 指令影响状态标志 AF、CF、OF、PF、SF 和 ZF。

例如：

```
SBB     AH,22                ; 字节减法
SBB     AX,BX                ; 字减法
SBB     AL,[BX]              ; 字节减法
SBB     VARW[BX],1234H       ; VARW 为字变量
SBB     20H[BX][SI],BL       ; 字节减法
```

【例 6.11】 假设(AL)=56H，(BL)=43H，CF=1。则指令“SBB AL,BL”执行结果是什么？

减法实际上是用加法做的。即把源操作数取反求补码，把 CF 也取反求补码（0 的补码为 0，−1 的补码为 1111 1111（8 位时）或 1111 1111 1111 1111（16 位时）），然后再做加法。指令执行结果是(AL)=13H，执行情况如下：

$$\begin{array}{r} (AL) = 56H = 01010110B \\ +\ \ -43A\text{的补码} = 00101111B \\ \hline 1\ \ 00010011B \qquad \text{有进位, 反而表示无借位, } CF = 0 \\ +\qquad 0\text{的补码} = 00000000B \\ \hline 00010011B \rightarrow AH \end{array}$$

3. 减 1 指令 DEC

减 1 指令 DEC 是将目的操作数内容减 1，结果存入目的操作数中，该指令主要用于调整地址指针和计数器，其格式如下。

格式：

```
DEC  DEST
```

其中，DEST 为目的操作数，可以是寄存器或存储器操作数，但不能是立即数和段寄存器操作数。

DEC 指令影响 OF、SF、ZF、PF 和 AF 标志位，但 CF 位不受影响（无论运算的实际结果是否有借位）。该指令常用在循环程序中，做循环次数计数器和对地址指针进行调整。

例如：

```
DEC     BX
DEC     SI
DEC     VARB            ; VARB 为字节变量
DEC     VARB[BX]        ; VARB 为字节变量
```

【例 6.12】　假设(AX)= 0D284H，则执行“DEC AX”指令后的结果是什么？

结果：

(AX)=(AX)-1=0D283H

注意：

对于无法确定类型的存储器操作数，必须用运算符 PTR 指定类型。

例如：

```
DEC     BYTE PTR 20H[BX][SI]
```

4. 求补码指令 NEG

求补码指令 NEG 也称为求负数指令，是将 0 减去目的操作数后送至目的操作数，使目的操作数符号变反，其格式如下：

格式：

```
NEG     DEST
```

其中，DEST 为目的操作数，可以是通用寄存器，也可以是存储单元，但不能为立即数。

NEG 指令影响状态标志 AF、CF、OF、PF、SF 和 ZF。要注意的是，除了操作数为 0 以外，CF 标志位总是 1；操作数为 0 时，CF 标志位置 0。

NEG 指令可以对字数据求负，也可以对字节数求负。

例如：

```
NEG     AL
NEG     AX
NEG     VARW[SI]                ; VARW 为字变量
```

下面就求补操作时对 CF 和 OF 两个标志位的影响做一个分析。求补操作即是把操作数按位求反后末位加 1，即 FFFFH−DEST+1= 0−DEST。也就是说，对一个操作数求补，相当于用 0 减去此操作数。参加求补运算的操作数当然是有符号数。对于 CF 而言，只有对 0 求补时，CF=0（无借位）；其余情况都使 CF=1（有借位）。0 减去负数应该是正数，但当操作数的值是 80H（−128）或 8000H(−32768)时，求补后其值保持不变，仍为负数，产生溢出，OF=l；其余情况都使 OF=0。

求反指令 NOT 的操作是对操作数按位求反，即 FFFFH−DEST。所以假定要对 AX 寄存器中的操作数进行求补，那么可以有如下几种方法。

方法 1：

```
NEG     AX
```

方法 2：

```
NOT     AX
INC     AX
```

方法 3：

```
MOV     BX,0
XCHG    AX,BX
SUB     AX,BX                    ；求补运算的结果应在 AX 中
```

【例 6.13】 假设(AX)= 0D284H，则执行“NEG AX”指令后的结果是什么？

结果：

(AX)=FFFFH−(AX)+1=2D7CH

5. 比较指令 CMP

比较指令 CMP 主要用于比较两个数的关系，是否相等，谁大，谁小。它将目的操作数内容减源操作数内容，结果不保存，目的操作数内容与源操作数内容保持不变，其格式如下。

格式：

```
CMPDEST,SRC
```

其中，

（1）DEST 为目的操作数，SRC 为源操作数，DEST 和 SRC 宽度必须一致，即同时为字数据或同时为字节数据。

（2）SRC 可以是通用寄存器、存储器或立即数，但 DEST 只能是通用寄存器或存储单元，不能是立即数，并且 DEST 和 SRC 不能同时为存储器操作数。

例如：

```
CMP     SI,DI
CMP     CL,2
CMP     AX,[BP]
```

比较指令 CMP 用于比较两个操作数的大小，根据比较的结果标志位，判断两个操作数的大小关系。该指令后面常跟条件转移指令，根据比较结果的不同产生不同的分支。它影响标志位 AF、OF、PF、SF、ZF 和 CF。

CMP 指令对标志寄存器各位的影响与 SUB 指令完全相同。根据执行 CMP 指令后 ZF 是否为 1 来判断两个操作数是否相等。如果两个操作数是无符号数，那么可以根据 CF 来判断两者大小关系，如果两个操作数是带符号数，那么可以根据 SF 和 OF 来判断两者的大小关系。

6.2.3 乘法指令

8086/8088 除了提供加减运算指令外，还提供乘除运算指令，乘除运算指令分为无符号数乘法指令和有符号数乘法指令，在这点上与加减运算指令不同，同时乘法指令对标志位的影响有点特别。

1. 无符号数乘法指令 MUL

无符号数乘法指令 MUL 的功能是如果是“字节”数据相乘，则将源操作数的内容乘以 AL 寄存器的内容，得到字数据结果送入 AX 寄存器中；如果是“字”数据相乘，则将源操作数的内容乘以 AX 寄存器的内容，得到双字数据结果，高字送入 DX 寄存器中，低字送入 AX 寄存器中，如图 6.8 所示。其格式如下。

格式：

```
MUL   SRC                    ; 被乘数隐含在累加器 AX（16 位）或 AL（8 位）中
```

其中，SRC 为源操作数。SRC 可以是寄存器操作数或存储器操作数，而不能是立即数和段寄存器。

例如：

```
MUL     BL                   ; 使(AX>=)(BL) ×(AL)
MUL     AX
MUL     VARW                 ; VARW 为字变量
```

MUL 指令只对 CF 和 OF 标志位有影响，如果乘积的高半部分（字节相乘是为 AH，字相乘是为 DX）不等于 0，则标志位 CF=1，OF=1；否则 CF=0，OF=0。所以如果 CF=1 和 OF=1 表示在 AH 或 DX 中含有结果的有效数，该指令对其他标志位无定义。

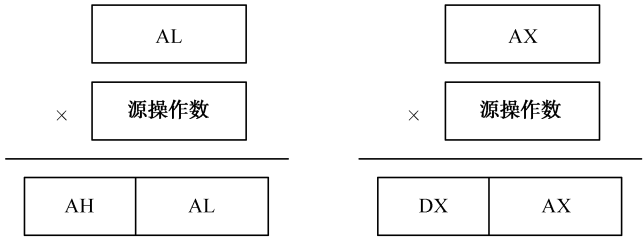


图 6.8　乘法运算示意图

【例 6.14】　设(AL)=2CH，(BL)= 11H，则执行如下指令后的结果是什么？

```
MUL   BL
```

结果：

使 2CH×11H，其乘积 02ECH 存储在（AX）中。

注意：

对于无法确定类型的存储器操作数，必须用运算符 PTR 指定类型。

2. 带符号数乘法指令 IMUL

带符号数乘法指令 IMUL 与 MUL 指令运算过程相同，只是操作对象为带符号的二进制数，其格式如下。

格式：

```
IMUL     SRC                 ; 被乘数隐含在累加器 AX(16 位)或 AL（8 位）中
```

其中，SRC 为源操作数。SRC 可以是寄存器操作数或存储器操作数，而不能是立即数和段寄存器。

例如：

```
IMUL     AL
```

```
IMUL    CX
IMUL    VARW                    ; VARW 为字变量
```

IMUL 指令只对 CF 和 OF 标志位有影响，如果乘积的高半部分（字节相乘是为 AH，字相乘是为 DX）不等于 0，则标志位 CF=1，OF=1；否则 CF=0，OF=0。所以如果 CF=1 和 OF=1，则表示 AH 或 DX 中含有有效数的结果，该指令对其他标志位无定义。

【例 6.15】 设(AL)= 0B4H，(BL)= 11H，则执行如下指令后的结果是什么？

```
IMUL   BL
```

结果：

(AX)=(0B4H) ×(11H)= 0FAF4H。

说明：

（1）计算二进制数乘法：B4H×11H。如果把它当做无符号数，用 MUL 指令结果为 0BF4H；如果看做有符号数，用 IMUL 指令则结果为 0FAF4H。由此可见，同样的二进制数看做无符号数与有符号数相乘，即采用 MUL 与 IMUL 指令，其结果是不同的。

（2）对于有符号数来讲，都是以补码形式来存储数据的。

6.2.4 除法指令

同乘法指令一样，除法指令也包括无符号数除法和有符号数除法。

1. 无符号数除法指令 DIV

无符号数除法指令 DIV 功能是如果是字节除法，就将 AX 寄存器的内容除以源操作数的内容，商值送 AL，余数送 AH；如果是字除法，就将 DX，AX 的内容除以源操作数的内容，商值送 AX，余数送 DX，其操作如图 6.9 所示。

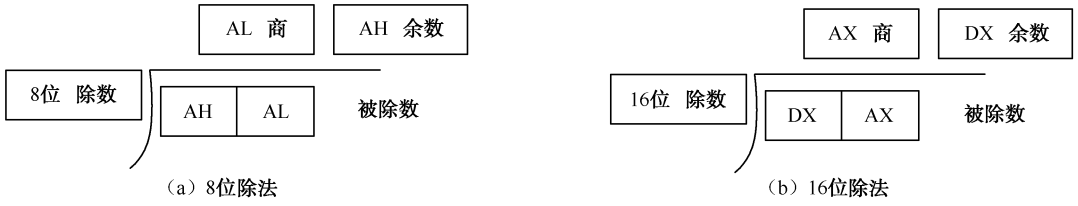


图 6.9 除法运算示意图

格式：

```
DIV     SRC             ; 被除数隐含在累加器 AH（16 位）中或两个寄存器 AX，DX 中
```

其中，SRC 为源操作数，可以是寄存器操作数或存储器操作数，而不能是立即数和段寄存器。

例如：

```
DIV     AL
DIV     CX
DIV     VARW                    ; VARW 为字变量
```

如果除数为 0，或者在 8 位数相除时商超过 8 位，或者在 16 位相除时商超过 16 位，则认为是除溢出，此时置 OF 标志位为 1，转入 0 号中断。

不允许两个字长相等的数相除。如果两个字长相等的数相除，可以在执行 DIV 指令前

将被除数向高位扩展，即将 AH 或 DX 清零。

DIV 指令对标志位的影响无定义。

【例 6.16】　设(AX)= 0400H，(BX)= 0B4H，则执行如下指令后的结果是什么？

```
DIV   BL
```

结果：

(AL)= 05H，(AH)= 7CH。

2. 有符号数除法指令 IDIV

有符号数除法指令 IDIV 与 DIV 指令相同，只不过各种数据都是带符号的。

格式：

```
IDIV    SRC          ; 被除数隐含在累加器 AX（16 位）或 AH 和 DX（共 32 位）中
```

其中，SRC 为源操作数，可以是寄存器操作数或存储器操作数，而不能是立即数和段寄存器。

例如：

```
IDIV    AL
IDIV    CX
IDIV    VARW              ; VARW 为字变量
```

使用 IDIV 指令时所用的操作数一定是有符号的，其数据都是以补码的形式存储的。而余数的符号和被除数的符号相同的。

不允许两个字长相等的数相除。如果两个字长相等的数相除，可以在执行 IDIV 指令前将被除数向高位扩展，即将 AL 或 AX 的最高位向 AH 或 DX 扩展，也就是将 AH 或 DX 清零（或“置 FFH 或 FFFFH”）。带符号数的扩展，可使用扩展指令 CBW 和 CWD。

执行 IDIV 指令时，如果除数为 0，或字节除法时商的值超出-128～+127 的范围，或字除法时商的值超出-32768 ～+32767 的范围，则 CPU 会立即自动产生一个类型号为 0 的内部中断，此时商和余数是不定值。

IDIV 指令对标志位的影响无定义。

【例 6.17】　设(AX)= 0400H，(BX)= 0B4H，则执行如下指令后的结果是什么？

```
IDIV   BX
```

结果：

(AL)= F3H，(AH)= 24H。

6.2.5　符号扩展指令

由于除法指令隐含使用字被除数或双字被除数，所以当被除数为字节，或者除数和被除数均为字时，需要在除操作前扩展被除数。8086/8088 专门提供了符号扩展指令。

1. 字节扩展为字指令 CBW

字节扩展为字指令 CBW 是将 AL 中的符号位数据扩展至 AH，其格式如下。

格式：

```
CBW
```

若 AL 中的符号位为 0，则扩展后(AH)= 00H；若为 1，则扩展后(AH)= FFH 。

例如：

```
MOV     AX,3487H
CBW
```

上例中第一条指令执行后 AX 为 3478H，即 AH 为 34H，AL 为 78H。第二条指令执行后 AX 为 0078H。

CBW 能在两个字节相除以前，产生一个字长度的被除数。不影响标志位。

2. 字扩展为双字指令 CWD

字扩展为双字指令 CWD 是将 AX 中的符号位数据扩展至 DX，其格式如下。

格式：

```
CWD
```

若 AX 中的符号位为 0，则扩展后(DX)= 0000H；若为 1，则扩展后(DX)= FFFFH，用法与 CBW 相同。

例如：

```
MOV     AX,4567H
CWD
```

上例中第一条指令执行后 AX 为 4567H。第二条指令执行后 AX 不变，DX 为 0。

注意：

在无符号数相除之前，不宜用 CBW 或 CWD 指令扩展符号位，一般使用 XOR 指令清高 8 位或高 16 位。

【例 6.18】 假设 X、Y 和 Z 均为 16 位带符号数，分别存放在名为 VARX、VARY 和 VARZ 的变量单元中。计算如下表达式的值，计算结果的商保存在 AX 中，余数保存在 DX 中。表达式如下：

```
(X*Y+Z-1000)/24
```

部分程序代码如下：

```
……
MOV     AX,VARX
IMUL    VARY                ; 计算 X×Y，结果保存在 DX、AX 中
MOV     CX,AX               ; 积的低 16 位保存到 CX 中
MOV     BX,DX               ; 积的低 16 位保存到 BX 中
MOV     AX,VARZ
CWD                         ; 把 VARZ 扩展成 32 位
ADD     AX,CX               ; 计算和
ADC     DX,BX
SUB     AX,1000             ; 计算差
SBB     DX,0
MOV     CX,24
IDIV    CX                  ; 计算商和余数，商保存在 AX 中，余数保存在 DX 中
……
```

6.2.6 BCD 码十进制调整指令

为了保证知识点的完整性和相互联系，BCD 码十进制调整指令在 10.2 节中介绍。

6.3　位操作类指令

位操作类指令是对字节或字中各位进行处理的指令。Intel 8086/8088 CPU 指令系统提供了三组位处理指令：逻辑运算指令、移位指令和循环移位指令。

6.3.1　逻辑运算指令

逻辑运算符 AND(与)、OR（或）、XOR（异或）和 NOT（非）作用是对操作数按位进行逻辑操作，它可以充当逻辑运算指令的“操作符”。逻辑运算指令用于完成对逻辑数据的运算，位与位之间没有进位或借位，没有正负和数值大小的区分。

1. 逻辑与指令 AND

逻辑与指令 AND 的功能是两个操作数（字节或字）进行按位“逻辑与”运算，结果送回目的操作数。若原来的两个操作数中的对应位均为 1，则结果中的该位置为 1；否则，该位置为 0。其格式如下。

格式：

```
AND DEST，SRC
```

其中，

（1）DEST 和 SRC 分别为目的操作数和源操作数。

（2）DEST 与 SRC 宽度一致，即同时为字数据或字节数据。

（3）SRC 可以是通用寄存器、存储单元或立即数。

（4）DEST 只能是通用寄存器或存储单元，不能是立即数。

（5）DEST 和 SRC 不能同时为存储器操作数。

AND 指令可以将一个字节或字中的某些特定位清零，或将某些特定位截取下来。AND 指令执行后，标志 CF=0，OF=0，PF、SF、ZF 反映操作的结果，AF 未定义。

当某个操作数自己与自己相“与”，则值不变，但可使进位标志 CF 清零。操作指令 AND 主要用在使一个操作数中的若干位维持不变，而另外若干位清为零的场合。把要维持不变的这些位与“1”相与，而把要清零的这些位与“0”相与就能达到这样的目的。

【例 6.19】 将寄存器 AL 设置为 34H，并将其中的内容高 4 位清零，低 4 位保存不变。

指令如下：

```
MOV     AL,34H                  ;  AL=34H
AND     AL,0FH                  ;  AL= 04H
```

说明：

例 6.19 中两条指令执行后 AL 为 04H，即 AL 高 4 位清 0，低 4 位保持不变，执行情况如下所示：

```
        (AL)   00110100B
∧       0FH    00001111B
-------------------------
        AL     00010011B
```

2. 测试指令 TEST

测试指令 TEST 用于对两个操作数进行逻辑与运算，但结果并不送至目的操作数，只是影响状态标志。此指令执行后两个操作数都不变。

格式：

```
TEST    DEST,SRC
```

其中，

（1）DEST 和 SRC 分别为目的操作数和源操作数。

（2）DEST 与 SRC 宽度一致，即同时为字数据或字节数据。

（3）SRC 可以是通用寄存器、存储单元或立即数。

（4）DEST 只能是通用寄存器或存储单元，不能是立即数。

（5）DEST 和 SRC 不能同时为存储器操作数。

指令 TEST 执行后，标志 CF=0，OF=0，PF、SF、ZF 反映操作的结果，AF 未定义。指令 TEST 常用来检测一些条件是否满足，但又不希望改变原有的操作数的情况，即适用于检查一个字节或字中的某些特定位是否有一位为 1。

例如：

要检查 AL 中的位 6 或位 2 是否有一位为 1，可使用如下的指令：

```
TEST    AL,01000100B
```

如果位 6 和位 2 全为 0，那么在执行上面的指令后，ZF 被置 1；否则 ZF 被清零。

通常是在 TEST 指令后面跟上一条条件转移指令，以便根据 ZF 是否为 0 来进行转移操作。

3. 逻辑或指令 OR

逻辑或指令 OR 将两个操作数（字节或字）进行按位逻辑“或”操作，并将结果送至目的操作数。如果两个操作数中对应位有一个为 1 或全为 1，则结果的该位为 1；否则，结果的该位为 0。其格式如下。

格式：

```
OR  DEST，SRC
```

其中，

（1）DEST 和 SRC 分别为目的操作数和源操作数。

（2）DEST 与 SRC 宽度一致，即同时为字数据或字节数据。

（3）SRC 可以是通用寄存器、存储单元或立即数。

（4）DEST 只能是通用寄存器或存储单元，不能是立即数。

（5）DEST 和 SRC 不能同时为存储器操作数。

例如：

```
OR      AX,8086
OR      BL,AL
OR      [BX],AX
```

指令 OR 执行后，标志 CF=0，OF=0，PF、SF、ZF 反映操作的结果，AF 未定义。当某个操作数自己与自己相或，则值不变，但可使进位标志 CF 清零。指令 OR 主要用在使一个操作数中的若干位维持不变，而另外若干位置为 1 的场合。把要维持不变的这些位与“0”相或，而把要置为 1 的这些位与“1”相或就能达到这样的目的。

【例 6.20】　将寄存器 AL 设置为 34H，并将其中的内容高 4 位置 1，低 4 位保存不变。指令如下：

```
MOV     AL,34H                          ; AL=34H
OR      AL,0F0H                         ; AL= F4H
```

说明：

例 6.20 中两条指令执行后 AL 为 F4H，即 AL 高 4 位置 1，低 4 位保持不变，执行情况如下所示：

$$\begin{array}{rl} & (AL)=00110100B \\ \vee & 0F0H=11110000B \\ \hline & 11110100B \rightarrow AL \end{array}$$

4. 逻辑异或指令 XOR

逻辑异或指令 XOR 将两个操作数（字节或字）进行按位逻辑异或操作，即两个操作数对应位不同，则结果对应位为 1；否则为 0，并将结果送至目的操作数，而源操作数内容保持不变。其格式如下。

格式：

```
XOR     DEST,SRC
```

其中，

（1）DEST 和 SRC 分别为目的操作数和源操作数。

（2）DEST 与 SRC 宽度一致，即同时为字数据或字节数据。

（3）SRC 可以是通用寄存器、存储单元或立即数。

（4）DEST 只能是通用寄存器或存储单元，不能是立即数。

（5）DEST 和 SRC 不能同时为存储器操作数

指令 XOR 对两个操作数进行按位的逻辑“异或”运算，运算结果送到目的操作数中。该指令执行以后，标志 CF=0，标志 OF=0，标志 PF、ZF、SF 反映运算结果，标志 AF 未定义。当操作数自己与自己相异或时，结果为 0，并可使进位标志 CF 清零。

例如：

```
XOR     DX,DX                           ; DX=0,   CF=0
```

异或操作指令主要用在使一个操作数中的若干位维持不变，而另外若干位取反的场合。把要维持不变的这些位与“0”相异或，而把要取反的这些位与“1”相异或就能达到这样的目的。

【例 6.21】　将寄存器 AL 设置为 34H，并将其中的内容高 4 位变反，低 4 位保存不变。指令如下：

```
MOV     AL,34H                          ; AL=34H
XOR     AL,0F0H                         ; AL= C4H
```

说明：

例 6.21 中两条指令执行后 AL 为 C4H，即 AL 高 4 位变反，低 4 位保持不变，执行情况如下所示：

$$\begin{array}{rl} & (AL)=00110100B \\ \oplus & 0F0H=11110000B \\ \hline & 11000100B \rightarrow AL \end{array}$$

5. 逻辑非指令 NOT

逻辑非指令 NOT 对目的操作数的每一位求反码，并将结果送回目的操作数。其格式如下。

格式：

```
NOT     DEST
```

其中，DEST 是目的操作数，可以是通用寄存器，也可以是存储单元，但不能是立即数。

例如：

```
NOT     AX
NOT     VARB                    ; VARB 是字节变量
```

指令 NOT 不改变状态标志。

【例 6.22】 将寄存器 AL 设置为 34H，并将其中的内容置反。

指令如下：

```
MOV     AL,34H                  ; AL=34H
NOT                             ; AL= CBH
```

6.3.2 移位指令

移位指令完成对数据的移位操作。每次默认移一位，当要移动多位时，规定移动位数必须放在寄存器 CL 中计数。移位分为算术移位和逻辑移位两种，而算术移位又包含算术左移和算术右移，逻辑移位包含逻辑左移和逻辑右移。

1. 逻辑左移指令 SHL

逻辑左移指令 SHL 将目的操作数左移 COUNT 次（位）。最高位移入进位标志 CF 中，而 CF 中原来的值被覆盖掉。移位后空出的最低位中填 0，如图 6.10 所示。其格式如下。

格式：

```
SHL     DEST,COUNT              ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。



图 6.10 逻辑左移 SHL 指令示意图

SHL 指令为单操作数指令，会根据移位后得到的结果修改标志寄存器 CF、PF、ZF、SF 和 OF，而 AF 无影响，CF 中总是包含从目的操作数中移出的最后一位的值。

若移位次数 COUNT 为 1，且移位前后目的操作数的最高位（即符号位）发生变化，那么 OF 置 1，否则 OF 清零；若移位次数 COUNT 大于 1，移位后 OF 的值则不能确定。

例如：

```
MOV     AL,2
SHL     AL,1                    ; 将(AL)左移一位
MOV     CL,3                    ; 设置移位位数
SHL     AL,CL                   ; 将(AL)左移 3 位
```

逻辑左移指令 SHL 可应用于“高位为 0 的无符号数”乘 2 操作（左移一位），以及乘 2^n（左移 n 位）的操作。通常，移位操作比乘法指令的操作快 10 倍以上。但要注意的是，一个数移动次数 n 次以后该数大小不能超过 2^N，这里 N 表示为被移动数的位数。同时还要注意，有溢出（高位的 0 移动完毕，出现将 1 移动到 CF 中）时，一定要改用乘法；否则，结果会出现错误。

2. 逻辑右移指令 SHR

逻辑右移指令 SHR 将目的操作数向右移 COUNT 次（位）。每移动一位，向高位补充一个 0，而移出的低位送入标志寄存器的 CF 位，如图 6.11 所示。

格式：

```
SHR     DEST,COUNT              ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。

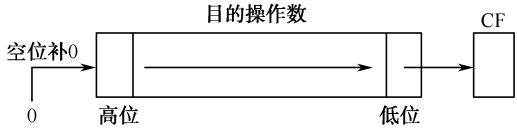


图 6.11　逻辑右移指令 SHR 示意图

指令 SHR 为单操作数指令，影响状态标志 CF、OF、PF、SF 和 ZF，但 AF 不确定。

若移位次数 COUNT 为 1，且移位前后目的操作数的最高位（即符号位）发生变化，那么 OF 置 1，否则 OF 清零。若移位次数 COUNT 大于 1，移位后 OF 的值则不能确定。

例如：

```
MOV     AL,2
SHR     AL,1                    ; 将(AL)右移一位
MOV     CL,3                    ; 设置移位位数
SHR     AL,CL                   ; 将(AL)右移 3 位
```

SHR 指令适用于将一个无符号数除以 2 或 2^n，右移 1 位等于将数除以 2。用 SHR 指令比用除法指令速度要快得多。另外，可以通过右移 1 位后看 CF 是否为 0，检查数是否为偶数。CF 为 1 表明此数是奇数。

逻辑移位指令还有一个很重要的用途，就是可以将一个字（或字节）中的某一位（或几位）移动到指定的位置，从而达到分离出这些位的目的。

【例 6.23】　将寄存器 AL 中的内容高 4 位与低 4 位互换。

指令如下：

```
MOV     BL,AL
MOV     CL,4                    ; 设置移位次数为 4
SHL     AL,CL                   ; 将低 4 位移至高 4 位，同时低 4 位都为 0
SHR     BL,CL                   ; 将高 4 位移至低 4 位，同时高 4 位都为 0
OR      AL,BL                   ; 执行或指令 OR 合并
```

3. 算术左移指令 SAL

算术左移指令 SAL 将目的操作数左移 COUNT 次（位）。最高位移入进位标志 CF 中，而 CF 中原来的值被覆盖掉。移位后空出的最低位中填 0，如图 6.12 所示。其格式如下。

格式：

```
SAL     DEST,COUNT              ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。

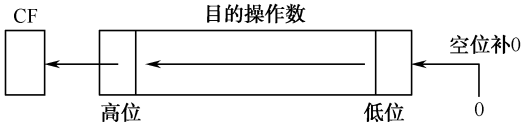


图 6.12　算术左移指令 SAL 示意图

SAL 指令与 SHL 指令完全相同，这里就不再叙述。

【例 6.24】　设 SI 为 00D3H，计算(SI)*8。

指令如下：

```
MOV     CL,3
SAL     SI,CL
```

说明：

结果 SI 为 0698H，无溢出。此方法要比乘法简单，但是若有溢出时，一定使用乘法。

4. 算术右移指令 SAR

算术右移指令 SAR 将目的操作数向右移 COUNT 次（位），每移动一位，向高位补充一个原有值，即最高位若为 1，则补充 1；若为 0，则补充 0，而移出的低位送入标志寄存器的 CF 位，如图 6.13 所示。

格式：

```
SAR     DEST,COUNT              ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。

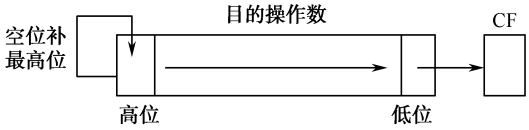


图 6.13　逻辑右移指令 SAR 示意图

指令 SAR 为单操作数指令，影响状态标志 CF、OF、PF、SF 和 ZF，但 AF 不确定。

若移位次数 COUNT 为 1，且移位前后目的操作数的最高位不会发生变化，那么 OF 置 1；否则 OF 清零。若移位次数 COUNT 大于 1，移位后 OF 的值则不能确定。

例如：

```
MOV     AL,2
SAR     AL,1                    ; 将(AL)右移一位
MOV     CL,3                    ; 设置移位位数
SAR     AL,CL                   ; 将(AL)右移 3 位
```

SAR 指令适用于将一个带符号数除以 2 或 2^n，右移 1 位等于将数除以 2。用 SAR 指令比用除法指令速度要快得多。

【例 6.25】　使用移位指令来计算−47/2。

指令如下：

```
MOV     AL,-47
SAR     AL,1
```

说明：

−47 的二进制码为 11010001B，所以当执行第二条指令时，算术右移 1 位为 11101000B，即-24，CF 位为 1。

6.3.3　循环移位指令

Intel 8086/8088 CUP 指令系统中有 4 条循环移位指令：不带进位标志的循环左移指令 ROL、不带进位标志的循环右移指令 ROR、带进位循环左移指令 RCL、带进位循环右移指令 RCR。这些指令可以一次只移一位，也可以一次移多位。如果移多位，那么移位次数存放在 CL 寄存器中。

所有循环移位指令只对标志位 CF 和 OF 有影响，但 OF 的含义对于左循环和右循环有所不同。

1．不带进位标志的循环左移指令 ROL

不带进位标志的循环左移指令 ROL 将目的操作数向左移 COUNT 位，每移动一位，把移出的最高位送入最低位和标志寄存器的 CF 位，如图 6.14 所示。其格式如下。

格式：

```
ROL     DEST,COUNT              ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT

的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。



图 6.14　不带进位标志的循环左移指令 ROL 示意图

指令 ROL 对标志位 CF 和 OF 有影响，如果移位一次后，操作数的最高位与原来的最高位(CF)不同，则记 OF=1；否则，记 OF=0。所以 OF 的值表示一次移位前后操作数的最高位有无变化。对于带符号数，则表示一次移位前后操作数的符号位有无变化。

2. 不带进位标志的循环右移指令 ROR

不带进位标志的循环右移指令 ROR 目的操作数向右移 COUNT 位，每移动一位，把移出的最低位送入最高位和标志寄存器的 CF 位，如图 6.15 所示。其格式如下。

格式：

```
        ROR     DEST,COUNT                    ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。



图 6.15　不带进位标志的循环右移指令 ROR 示意图

循环右移指令 ROR 对标志位 CF 和 OF 有影响，如果移位一次后，操作数的最高位与原来的最高位(CF)不同，则记 OF=1；否则，记 OF=0。所以 OF 的值表示一次移位前后操作数的最高位有无变化。对于带符号数，则表示一次移位前后操作数的符号位有无变化。

3. 带进位循环左移指令 RCL

带进位循环左移指令 RCL 将目的操作数向左移 COUNT 位，每移动一位，把标志寄存器的 CF 位送入最低位，再把移出的最高位送入 CF 位，如图 6.16 所示。其格式如下。

格式：

```
        RCL     DEST,COUNT                    ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT

的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。

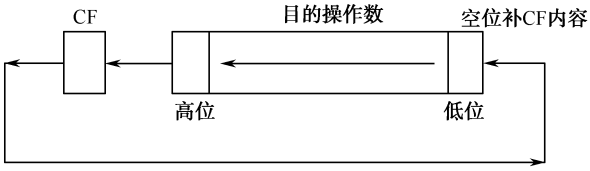


图 6.16　带进位循环左移指令 RCL 示意图

带进位循环左移指令 RCL 对标志位 CF 和 OF 有影响，与指令 ROL 相同。

4. 带进位循环右移指令 RCR

带进位循环右移指令 RCR 是将目的操作数向右移 COUNT 位，每移动一位，把标志寄存器的 CF 位送入最高位，再把移出的最低位送入 CF 位，如图 6.17 所示。其格式如下。

格式：

```
        RCR     DEST,COUNT                    ; COUNT=1 或 CL
```

其中，

（1）DEST 为目的操作数，可以是通用寄存器，也可以是存储器；DEST 可以是字数据，也可以是字节数据。

（2）COUNT 为移动次数，如果 COUNT=1，指令中可以直接使用常数 1 来代替 COUNT 的位置；如果 COUNT>1，那么必须由寄存器 CL 来代替 COUNT 在指令中的位置。

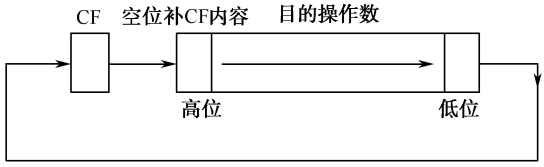


图 6.17　带进位循环右移指令 RCR 示意图

带进位循环右移指令 RCR 对标志位 CF 和 OF 有影响，与指令 ROR 相同。

注意：

（1）对于不带进位标志的循环移位指令 ROL 和 ROR 来说，如果操作数是 8 位，那么在移位 8 次后操作数就能复原，如果操作数位 16 位，那么在移动 16 位后操作数就能复原。

（2）对于带进位循环移位指令 RCL 和 RCR 来说，如果操作数是 8 位，那么在移位 9 次后操作数就能复原，如果操作数位 16 位，那么在移动 17 位后操作数就能复原。

【例 6.26】　把 AL 最低位送 BL 最高位，但保持 AL 不变。

指令如下：

```
        ROR     BL,1
        ROR     AL,1
        RCL     BL,1
        ROL     AL,1
```

6.4 处理器控制类指令

Intel 8086/8088 CPU 的指令系统中还提供了用于控制处理器状态的指令，可以分为两大类，一类是标志位操作指令，用于修改标志寄存器各位的状态；另一类是 CPU 控制指令，用于控制 CPU 的状态。

6.4.1 标志位操作指令

8086 指令系统中，有 3 条直接针对进位标志 CF，2 条直接针对方向标志 DF，2 条直接针对中断标志 IF 的操作指令，分别介绍如下。

1. 进位标志清零指令 CLC

进位标志清零指令 CLC 将进位位 CF 清零，即 CF=0。其格式如下。

格式：

```
CLC
```

2. 进位标志置位指令 STC

进位标志置位指令 STF 将进位位 CF 置 1，即 CF=1。其格式如下。

格式：

```
STC
```

3. 进位标志取反指令 CMC

进位标志取反指令 CMC 将进位位 CF 变反，即 CF = $\overline{CF}$。其格式如下。

格式：

```
CMC
```

4. 方向标志清零指令 CLD

方向标志清零指令 CLD 将方向位 DF 清零，即 DF=0。其格式如下。

格式：

```
CLD
```

5. 方向标志置位指令 STD

方向标志置位指令 STD 将方向位 DF 置 1，即 DF=1。其格式如下。

格式：

```
STD
```

6. 中断标志清零指令 CLI

中断标志清零指令 CLI 将中断允许位 IF 清零，即 IF=0。其格式如下。

格式：

```
CLI
```

7. 中断标志置位指令 STI

中断标志置位指令 STI 将中断允许位 IF 置 1，即 IF=1。其格式如下。

格式：

```
STI
```

6.4.2　CPU 控制指令

CPU 控制指令用于控制 CPU 的状态，它们都不会对标志寄存器各位产生影响。

1. 空操作指令 NOP

空操作指令 NOP 也称为无操作指令，它不执行任何操作，其机器码占有一个字节，在调试程序时往往用这条指令占有一定的存储单元，以便在正式运行时用其他指令取代。NOP 空操作指令还可用来达到精确的时间延迟。其格式如下。

格式：

```
NOP
```

2. 停机指令 HLT

停机指令 HLT 可使机器暂停工作，使 CPU 处于停机状态，以便等待一次外部（硬件）中断到来。中断结束后，CPU 会退出暂停状态，继续执行下面的程序。系统复位操作也会使 CPU 退出停机状态。其格式如下。

格式：

```
HLT
```

指令 HLT 不影响任何标志位，在程序必须等待中断的情况下，可用 HLT 指令代替无限的软件循环，同时还可用于多机系统的同步。

3. 等待指令 WAIT

等待指令 WAIT 指令使 CPU 处于空转状态，它也可以用来等待外部中断的发生，但中断结束后仍返回 WAIT 指令继续运行。其格式如下。

格式：

```
WAIT
```

在程序设计中，WAIT 指令可用于使 CPU 与协处理器或外部设备同步工作。

4. 放权指令 ESC

放权指令 ESC 把指定存储单元中的内容送到数据总线上，从而将存储器中的指令或数据传送给其他处理器。其格式如下。

格式：

```
ESC     OPRD, SRC
```

其中，

（1）OPRD 为外部操作码，是一个 6 位立即数，它指明了其他处理器（如协处理器）要完成的功能。

（2）SRC 为源操作数，可以是通用寄存器，也可以是存储单元。

ESC 指令提供了一种机制，使得其他处理器（如协处理器等）可以通过 ESC 接收 8086/8088 的指令流。

5. 封锁指令 LOCK

封锁指令 LOCK 是一种前缀，它可与其他指令联合，用来维持总线的锁存信号直到与其联合的指令执行完为止（独占直线）。其格式如下。

格式：

```
LOCK    OP
```

其中，OP 为其他指令。

当 CPU 访问存储器和外设时，该指令可对总线锁存，避免其他处理器获取总线控制权，以协调其他处理机的工作，避免破坏有用信息。

习题

1．指出下列指令哪些是错误的，并说明原因。

（1）MOV AX,BL
（2）MOV CS,AX
（3）MOV 80H,AL
（4）MOV IP,AX
（5）MOV AX,[SI]
（6）MOV [DI],BL
（7）MOV [SI], [DI]
（8）PUSH CH
（9）POP CS
（10）ADD ES,AX
（11）INC [AX]
（12）CMP AL,2467H
（13）SUB 1234H,AX
（14）NOT AX,CL
（15）RCR DX,3
（16）LEA DS,123H[SI]

2．写出完成以下功能的指令。

（1）加 24H 到 CX。
（2）把 123 存入 CX。
（3）交换字变量 VARW 和 BX 的内容。
（4）清除 DH 中最低 3 位而不改变其他位，结果存入 BH。
（5）AL 右移一位，将 0 移入最低位。
（6）将 AL 中的压缩 BCD 码变成 AX 中的非压缩 BCD 码。
（7）将 DX 右移 5 位且保持正负特征不变。
（8）将 DI 的高 5 位置 1，其他位保持不变。
（9）将 DX 的第 0～4 位置 1，第 6～8 位取反，第 13～15 位清 0，其他位保持不变。
（10）将标志寄存器中的 IF 位置 1。

3．BUF 是数据段中定义的字类型的变量，BUF 的有效地址为 0020H，存放内容为 ab，试说明下面两条指令的区别，并且指出指令执行后 DX 寄存器的内容是什么？

（1）MOV DX,BUF
（2）LEA DX,BUF

4．简述 XLAT 指令的功能及使用方法。

5．比较 CMP 指令与 SUB 指令的异同点。

6．编写程序段，将 AL、BL、CL、DL 相加，结果存放在 DX 中。

7．设寄存器 AX 中存放 1234H，寄存器 BX 中存放 1，执行“DIV BL”指令后，会出现什么现象？为什么？

8．假设(DX)=347DH，(CL)=5，CF=0，执行下列指令后，DX 和 CF 中的值是什么？

（1）SHL　DH,1

（2）SAL　DX,CL

（3）SHR　DL,1

（4）SAR　DL,1

（5）ROL　DX,CL

（6）RCR　DL,1

（7）RCLDH,CL

9．在程序中数据段的定义如下：

```
VAR1    DW  ?
VAR2    DB      1,2,3,4,5
VAR3    DW  100 DUP (?)
```

（1）编写指令，使用两种方法将 VAR1 的有效地址送 AX 寄存器。

（2）编写指令，将 VAR2 中第一个字的内容保存到 AX 寄存器。

（3）编写指令，将 AX 寄存器的内容保存到 VAR3 中第一个字的位置。

10．执行下面两条指令后，BX 寄存器的内容是什么？执行前(BX)=4D6EH。

```
MOV     CL,7
SHR     BX,CL
```

11．编写程序段，要求把 BL 中的数除以 CL 中的数，并把商乘以 3，运算的结果存放在 DX 寄存器中。

12．编写程序，在数据段中定义两个字节类型的数组变量 buff1 和 buff2，使用 MOV 指令将 buff1 中的字符传送到 buff2 中。

13．编写程序，在数据段中定义两个相同大小的字节类型的数组变量 buff1 和 buff2，使用 XCHG 指令交换这两个变量的内容。

第 7 章

基本程序设计

编写一个结构合理、快速高效和稳定可靠的大型程序是一件很困难的事，必须经过长期的编程训练才能达到。而汇编语言是最接近计算机核心的编码语言，编写高质量的程序，可以最大限度地发挥计算机硬件的性能。

同其他程序设计语言一样，汇编语言基本的程序设计结构有顺序结构、分支结构、循环结构、子程序等几种形式，它们是汇编语言程序设计的基础。本章重点介绍顺序结构、分支结构和循环结构的程序设计方法。

7.1 程序设计方法

程序是解决某个问题的指令或语句的有序结合。使用某种计算机语言，编写解决某一问题的程序的过程称为程序设计。汇编语言程序设计是用计算机的汇编语言指令、伪指令和宏指令编写解决某个问题的程序过程。

7.1.1 程序设计步骤

一般说来，编制一个汇编语言程序，与编制其他高级语言程序类似，一般需要按照下述几个步骤进行。

1. 分析问题

分析问题的目的就是要对问题有一个确切的理解，明确问题的环境限制，弄清已知条件、原始数据、输入与输出要求、对运算精度的要求、对处理速度的要求，以及最后期望获得的结果。正确地分析问题是进行程序设计的基础。

2. 确定解决问题的合理算法

汇编程序算法是指用汇编语言在有限步之内求解某一类问题的方法与步骤，它可能依据某一数学模型，也可能直接或间接采用一些现有的计算方法。开始时一般先模拟计算机解决实际问题的过程，根据算法把一个实际问题转化为一个计算机可以处理的问题。在考虑算法时，如果选择不同的输入或输出数据的方法、格式、数据的表示方法，那么对于同一个问题可能有多种算法，这时还应根据问题的要求，选择较优的算法。

3. 绘制程序流程图

流程图是用框图的形式来表示解题的方法、步骤。流程图不仅描述算法清晰、直观，并且可以减少程序逻辑出错的可能性。如果要解决的问题比较复杂，可以先编写比较大的过程框图构成的程序流程图，再逐步细化，直到每一框图都可以很容易编制程序为止。在设计时要注意程序的结构应尽可能简单，层次尽可能清晰。

4. 分配存储空间和工作单元

与高级语言程序不同，汇编语言程序要求运行的每一步都能清楚数据的位置，因此，在程序设计过程中，始终要关心如何选用存储空间与如何选择寄存器的问题。一个完整的汇编程序往往由一个或几个段构成，要分别定义程序中所需使用的数据段、堆栈段、代码段及附加段。要根据程序执行过程合理地对每一步及各个阶段所使用的工作单元进行安排，工作单元既可以是存储单元，也可以是数据寄存器。在设计时要注意，占用内存应当尽可能少，占用内存的情况由存储单元的分配及程序流程情况共同决定。

5. 根据程序流程图编写程序

在完成以上准备工作后，就可以根据程序流程图用汇编语句实现算法，这一过程称为编制程序，或称为编码。在编制程序时，必须严格按照汇编语言的语句规则与语法规则书写，还要注意保证源程序的可读性和可维护性。用汇编语句编写出来的程序称为汇编语言源程序，再经过汇编后变为机器语言程序，再经过连接成为可执行程序。

6. 静态检查及上机调试

前面步骤仅仅完成源程序的编写，但如何才能检查出设计思想是否正确，以及程序逻辑是否符合设计？这一切都只有静态检查和进行调试运行后才能完成。首先要进行静态检查，看程序是否具有所要求的功能，程序是否清晰易读，选用的指令是否合适，程序的语法和格式上是否有错，等等。静态检查认为无错了，就可以在计算机上进行编译、连接与调试。也只有在计算机上运行通过的程序，才能确认是正确的程序。上机调试程序大致有以下 4 个步骤。

（1）使用编辑程序，输入源程序，并保存成扩展名为.ASM 的文件。

（2）使用宏汇编程序，把扩展名为.ASM 的源程序汇编成目标程序，即生成扩展名为.OBJ 的文件。

（3）使用连接程序，把扩展名为.OBJ 的目标程序连接装配成可执行文件，即生成扩展名为.EXE 的文件。

（4）使用调试程序，调试扩展名为.EXE 的可执行文件。

7.1.2　程序流程图

程序流程图主要由以下几种框图符号组成，框图表示如图 7.1 所示。

1. 流向线

流向线是用箭头线表示程序的流向。流向线表示程序执行的顺序关系。如果程序的流向是从上往下或从左往右，有时可以不画箭头，否则必需用箭头指明程序的流向。

2. 起止框

起止框是用带圆角的方框表示一个程序或一个程序模块的开始和结束。起始框内通常

用程序名、过程名、标号或“开始”字符来表示，它仅有一个出口。终止框内通常用“结束”、“返回”等字符来表示，它仅有一个入口。

3. 判断框

判断框是用菱形表示程序的分支与流向，框内记录判断条件。这种框图有一个入口，两个出口。在每个出口上要注明分支流向条件。

4. 处理框

处理框是用矩形说明一段程序（或一条指令）所完成的功能。这种框图通常是一个入口，一个出口。

5. 连接框

连接框是用小圆圈表示程序段之间的关系。当一个程序比较复杂，需要分布在几张纸上，或者虽然在一张纸上，但是流程图中连线较多且纵横交错时，可用连接框表示两根流向线间的连接关系。在连接框中常用字母或数字来表示谁和谁连接，两个框内有相同字母或数字的连接框表示它们是连在一起的。它只有一个入口或出口。

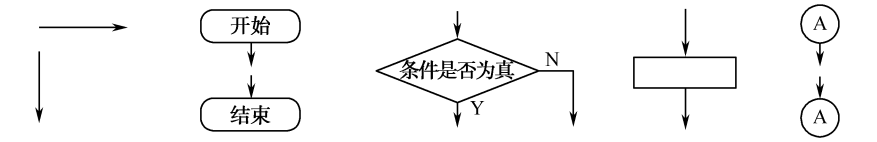


图 7.1　程序流程图的框图符号

7.2　顺序结构程序设计

顺序结构是最简单的程序结构，程序的执行顺序就是指令的编写顺序，所以，安排指令的先后次序就显得至关重要。另外，在编写程序时，还要妥善保存已得到的处理结果，后面步骤可直接采用前面的处理结果，从而避免不必要的重复操作。

7.2.1　顺序程序设计

因为顺序程序设计比较简单，下面通过实例进行说明。

【例 7.1】　试写出两个字节数据相加的程序。

（1）分析问题

两个字节数据相加比较简单，可以直接使用加法指令。当然首先要定义 2 个字节变量 X 和 Y，再定义字变量 Z 用于存放相加后的结果，这里假设 X 为 3CH，Y 为 94H。

（2）确定算法

首先将 X 读到 AL 中，再和 Y 相加，同时 AH 清零后和进位标志位相加，则 AX 为相加的结果，再将 AX 存储到 Y 中。

（3）画程序流程图

流程图如图 7.2 所示。

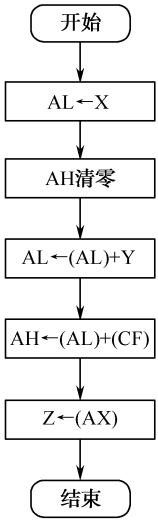


图 7.2　两个字节数相加的流程图

（4）确定程序的基本框架

该汇编语言程序的基本框架至少要两个段，即数据段和代码段。数据段中至少定义 3 个变量，即两个加数变量 X、Y，都是 DB 类型，还有一个和变量 Z，是 DW 类型。和变量 Z 尚属未知，可以定义一个 1 个字长的缓冲区。

（5）编写程序

程序如下：

```
DATA    SEGMENT                    ; 定义数据段
X       DB   3CH                   ; 定义字节变量 X
Y       DB   94H                   ; 定义字节变量 Y
Z       DW   ?                     ; 用于保存结果
DATA    ENDS                       ; 数据段结束
CODE    SEGMENT                    ; 定义代码段
        ASSUME  CS:CODE,DS:DATA    ; 指定段寄存器
START:  MOV     AX,DATA
        MOV     DS,AX              ; 把数据段的段首址赋给段寄存器 DS
        MOV     AL,X               ; X 送入 AL 中
        MOV     AH,0               ; H 清零
        ADD     AL,Y               ; 和 Y 相加
        ADC     AH,0               ; AH 加上进位
        MOV     Z,AX               ; 将和送入 Z 中
        MOV     AH,4CH
        INT     21H
CODE    ENDS                       ; 代码段结束
        END     START              ; 程序结束
```

【例 7.2】　试写出两个 64 位无符号数相加的程序。

（1）分析问题

在 Intel 8086/8088 CPU 指令系统中，只有 8 位或 16 位运算指令，没有 32 位和 64 位

以上的运算指令。

（2）确定算法

要进行 64 位的加法运算，可以利用 16 位加法指令分别相加 4 次来实现。

（3）画程序流程图

流程图如图 7.3 所示。

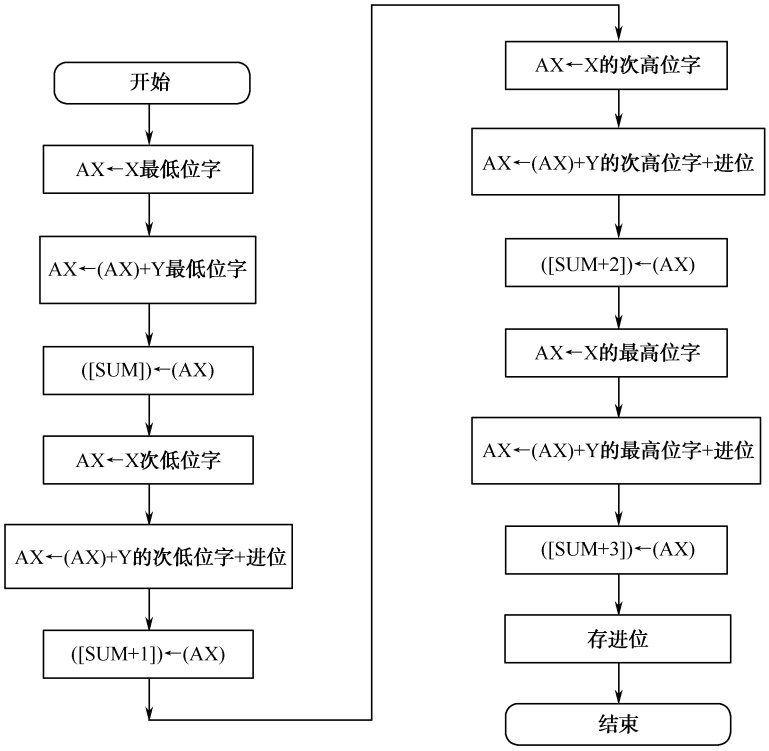


图 7.3　两个 64 位无符号位数相加的流程图

（4）确定程序的基本框架

该汇编语言程序的基本框架至少要两个段，即数据段和代码段。数据段中至少定义 3 个变量，即两个加数变量 X、Y，还有一个和变量 SUM，都是 DW 类型。和变量 SUM 尚属未知，可以定义一个 5 个字长（比 64 位字长大一点）的缓冲区。因为要加 4 次，需要一个计数器或指针，可选择寄存器 BX。

（5）编写程序

程序如下：

```
DATA    SEGMENT                                  ; 定义数据段
X       DW   2434H, 0E648H, 98BCH, 0D0F0H        ; 定义字变量 X
Y       DW   3751H, 7314H, 9978H, 3163H          ; 定义字变量 Y
SUM     DW   5DUP(?)                             ; 定义缓冲区 SUM
DATA    ENDS                                     ; 数据段结束
CODE    SEGMENT                                  ; 定义代码段
        ASSUME  CS:CODE, DS:DATA                 ; 指定段寄存器
START:  MOV     AX,DATA                          ; 指令开始
```

```
        MOV     DS,AX           ; 把数据段的段首址赋给段寄存器 DS
        LEA     BX,X            ; 将 X 的偏移地址送入 BX 中
        MOV     AX,[BX]         ; 引用 DS 来访问 DATA 中的变量 X 的最低位
        ADD     AX,[BX+8]       ; 最低位字相加，两数偏移地址相差 8 个字节
        MOV     [SUM],AX        ; 存最低位和。
        INC     BX
        INC     BX              ; 指针指向下一个字
        MOV     AX,[BX]         ; 引用 DS 来访问 DATA 中的变量 X 的次低位
        ADC     AX,[BX+8]       ; 次低位字相加，两个数偏移地址相差 8 个字节
        MOV     [SUM+1],AX      ; 存次低位和。
        INC     BX              ; BX←BX+1
        INC     BX              ; BX←BX+1，指针指向下一个字
        MOV     AX,[BX]         ; 引用 DS 来访问 DATA 中的变量 X 的次高位
        ADC     AX,[BX+8]       ; 次高位字相加，两数偏移地址相差 8 个字节
        MOV     [SUM+2],AX      ; 存次高位和。
        INC     BX              ; BX←BX+1
        INC     BX              ; BX←BX+1，指针指向下一个字
        MOV     AX,[BX]         ; 引用 DS 来访问 DATA 中的变量 X 的最高位
        ADC     AX,[BX+8]       ; 最高位字相加，两数偏移地址相差 8 个字节
        MOV     [SUM+3],AX      ; 存最高位和。
        MOV     AX,0            ; 对 AX 清零
        ADC     AX,0            ; 计算进位
        MOV     [SUM+4],AX      ; 存进位位
        MOV     AH,4CH          ; 返回 DOS 系统
        INT     21H
CODE    ENDS                    ; 代码段结束
        END     START           ; 程序结束
```

7.2.2　简单查表法代码转换

在汇编语言程序设计中，会经常用到代码转化。例如，将 BCD 码转化为 ASCII 码就是很好的例子。对于各种不同代码之间的转换，往往要采用各种不同的方法，以便获得最佳的效率。查表是实现代码转换的一种方法。

【例 7.3】　编写一个能把十六进制数字码（0，…，9，A，b，c，d，E，F）转换为对应七段代码的程序。

（1）分析问题

七段显示数码管（如图 7.4 所示）的管 1～7 对应一个字节数据的第 0～6 位，如果假设 0 表示对应段亮，1 表示对应段暗，那么字节数据的第 7 位无效。假设恒置 0，那么数字码 0 对应以二进制形式表示的代码 01000000，数字码 1 对应以二进制形式表示的代码 01111001，这样，数字码 F 对应以二进制形式表示的代码 00001110。这种用于表示七段数码管亮暗的代码就称为七段代码。七段代码可以在七段显示数码管中能较好地显示十六进制数字。

（2）确定算法

因为十六进制数字码与七段代码间的关系难以表示成一个简单的算术表达式，所以利用表的方法实现代码转换较为合适。首先把十六进制数字码（0，…，9，A，b，c，d，E，F）对应的七段代码制成表格，使用查表指令查找出十六进制数据码所对应的七段代码。

（3）画程序流程图

流程图如图 7.5 所示：

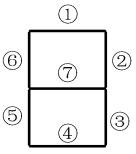


图 7.4　七段显示数码管

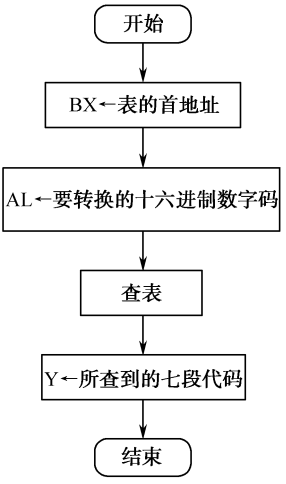


图 7.5　流程图

（4）确定程序的基本框架

该汇编语言程序的基本框架至少要两个段，即数据段和代码段。在数据段中定义一个表 TAB，按（0，…，9，A，b，c，d，E，F）存放对应的十六进制数据码所对应的七段代码，再定义要转换的十六进制数字码 X 和存放对应的七段代码 Y，都是 DB 类型。

（5）编写程序

程序如下：

```
DATA    SEGMENT                          ; 定义数据段
TAB     DB  01000000B,  01111001B,  00100100B,  00110000B  ; 七段代码表
        DB  00011001B,  00010010B,  00000010B  01111000B
        DB  00000000B,  00010000B,  00001000B,  00000011B
        DB  0l000ll0B,    00l00001B,  00000110B,  00001110B
X       DB  5                            ; 要转换的十六进制数字码
Y       DB  ?                            ; 存放对应的七段代码
DATA    ENDS                             ; 数据段结束
CODE    SECMENT                          ; 定义代码段
ASSUME  CS: CODE ,DS: DATA               ; 指定段寄存器
START:  MOV     AX,DATA                  ; 指令开始
        MOV     DS,AX                    ; 把数据段的段首址赋给段寄存器 DS
        MOV     BX,OFFSET TAB            ; 将表的首地址存到 BX 中
        MOV     AL,X                     ; 将要转换的十六进制数字码送入 AL 中
        AND     AL,0FH                   ; 保证在 0 至 F 之间
        XLAT                             ; 查表
```

```
        MOV     Y,AL                ; 保存所查到的七段代码
        MOV     AH,4CH              ; 返回 DOS 系统
        INT     21H
CODE    ENDS                        ; 代码段结束
        END     START               ; 程序结束
```

说明：

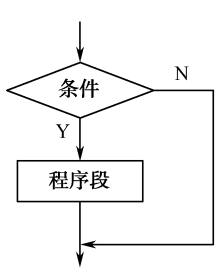
指令 XLAT 把寄存器 BX 的内容作为表（每一项一个字节）的首地址，把寄存器 AL 的值作为下标，取出表项内容再送入 AL 寄存器。也就是把寄存器 AL 中的内容转换成表中对应项的值，此即所谓的换码。使用此指令前，应先把表的首地址送 BX 寄存器。表最大为 256 项。此指令属于数据传送指令组。

7.3　分支程序设计

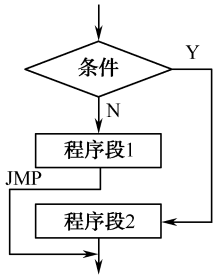
在许多实际问题中，往往需要对不同的情况或条件做出不同的处理，这就要用到分支结构程序。分支结构是实现程序选择功能所必需的程序结构，它主要通过转移指令来实现。转移指令包含无条件转移指令和条件转移指令。

分支程序根据其支路的多少，可分为单分支程序、双分支程序和多分支程序三类。

单分支程序根据条件成立与否，执行某一程序段或跳过该程序段，如图 7.6（a）所示，其中 Y 表示条件成立，N 表示条件不成立，它类似于高级语言中的“IF THEN”语句；双分支程序根据条件成立与否分别执行程序段 1 或程序段 2，如图 7.6（b）所示，它类似于高级语言中的“IF THEN ELSE”语句。



（a）单分支程序结构示意图



（b）双分支程序结构示意图

图 7.6　单和双分支程序结构示意图

多路分支结构如同高级语言中的“CASE”语句，常常根据各种不同的情况执行多路转移。在指定范围内，每一次条件测试都将执行一路程序，如图 7.7 所示。

条件转移指令和无条件转移指令用于实现程序的分支。条件转移指令是实现分支结构最常用和最灵活的指令，该指令可根据当前某些标志位的状态实现转移或不转移。因此，在条件转移指令前，通常需要安排算术运算指令、比较指令或测试指令等能够影响标志位的相关指令。无条件转移指令 JMP 指令在分支程序中作为辅助性的指令，实现固定转移到某个指定的位置。

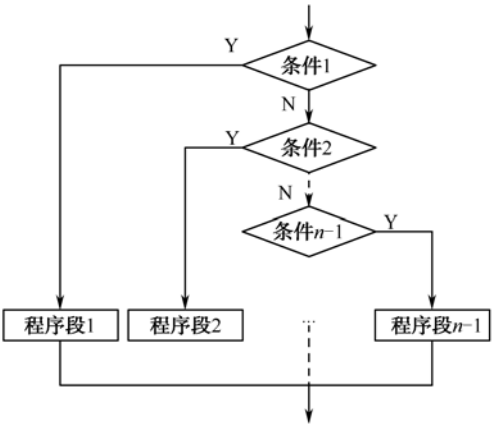


图 7.7　多路分支程序结构示意图

7.3.1　无条件转移指令

无条件转移指令 JMP 使程序无条件地转移到指令指定的地址去执行。在 JMP 指令中，必须指定转移的目标地址（或称为转移地址）。比如，通过改变 IP 的数值，让程序从当前位置跳到同一段的其他位置继续执行，称为“段内转移”。同时改变 CS 和 IP 的数值，让程序从当前代码段跳到其他代码段中的另一位置继续执行，称为“段间转移”。

JMP 指令的执行不影响标志寄存器的各位，它有以下几种指令格式。

1. 段内直接短转移

段内直接短转移是指从当前指令跳转到同一代码段的另一指令（目标标号）处，转移量不超过 8 位，即距离当前 IP 位置的−128～+127 字节之间。其指令格式如下。

格式：

```
JMP     SHORT   TARGET
```

其中，TARGET 表示的是目标标号。

段内直接短转移执行的操作为 (IP)←(IP)+“TARGET 与 JMP 指令的下一条指令的 8 位位移量距离”。

例如：

```
        …
NEXT:   MOV     AX,BX
        …
        JMP     SHORT   NEXT            ; 转到 NEXT 处
        …
        JMP     SHORT   HELLO           ; 转到 HELLO 处
        …
HELLO:  MOV     AX,67H
        …
```

此处目标标号 NEXT 和 HELLO 语句离对应的 JMP 语句的范围在-126~129 字节之间。只需改变 IP 寄存器的内容，即用新的转移目标地址代替原有的 IP 的值，就可达到转

移的目的。JMP 相应的机器指令占用两个字节，因此指令跳转的地址的起点应该是 JMP 指令的下一条指令的地址。目标标号所指示地址与当前 IP 寄存器的内容（即 JMP 指令的下一条指令的地址）的差，称为指令的转移量 DISP。它由汇编程序或编译程序根据目标地址和转移指令的距离计算得出，汇编指令中只需给出目标标号即可。转移地址的偏移地址应该是当前 IP 寄存器的内容（是 JMP 指令的下一条指令的地址）和机器指令中指定的转移量之和，那么，转移地址的偏移地址 EA 与 IP 和 DISP 之间的关系为：

EA=IP+ DIS

【例 7.4】 设指令“JMP　SHORT　NEXT”的偏移地址为 1254H，NEXT 为目标标号，其偏移地址为 12D0H。试问汇编程序对该指令进行汇编后，指令机器码中给出的转移量是多少？

由于 JMP 指令的偏移地址为 1254H，它的机器码是两个字节，因此它的下一条指令的首地址（在 IP 中）应该为：

IP=1254H+2=1256H

JMP　SHORT　NEXT 的执行示意图如图 7.8 所示。

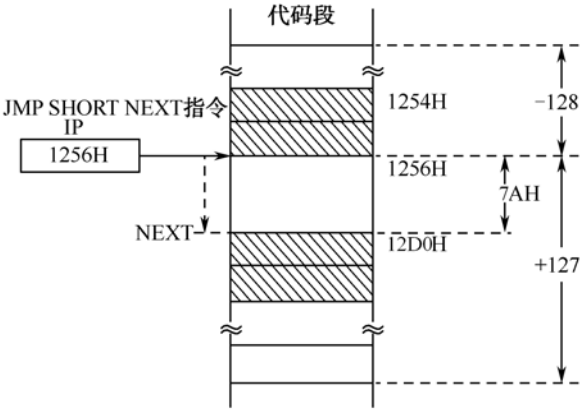


图 7.8　JMP SHORT TABL 执行示意图

转移地址的偏移地址 EA 与 IP 和 DISP 之间的关系为：

EA=IP+ DISP

现在，已知 EA=12D0H，IP=1256H，则 DISP 的值为：

DISP=EA−IP =7AH

故指令机器码中给出的转移量是 7AH。

2. 段内直接近转移

段内直接近转移与段内直接短转移基本相同，只是转移量为 16 位，即−32768～+32767 字节。该指令相应的机器指令占用 3 个字节，因此 JMP 指令的下一条指令的地址应该等于 JMP 指令的地址加上 3。其格式如下。

格式：

```
JMP　　NEAR　PTR　TARGET
```

其中，

（1）TARGET 表示的是目标标号。

（2）NEAR　PTR 也可以不用。

段内直接近转移执行的操作为 (IP)←(IP)+“TARGET 与 JMP 指令的下一条指令的 16 位位移量”。

例如：

```
            …
NEXT:       MOV     AX,BX
            …
            JMP     NEAR  PTR   NEXT          ; 转到 NEXT 处
            …
            JMP     HELLO                     ; 转到 HELLO 处，省略了 NEAR PTR
            …
HELLO:      MOV     AX,67H
            …
```

此处目标标号 NEXT 和 HELLO 语句离对应的 JMP 语句的范围在-32766~32769 字节之间。

注意：

（1）汇编程序有可能会根据转移的距离对段内直接近转移进行优化，将其汇编成段内直接短转移指令。

（2）无论是 8 位，还是 16 位，转移量为负时，向低地址跳转；为正时，向高地址跳转。

3. 段内间接转移

段内间接转移跳转的目标地址不直接给出，而是由 TARGET 的寻址方式（寄存器或存储单元的内容）或伪指令 DW 定义的字变量（符号地址）确定的偏移地址 EA 确定，然后执行该地址中的相关指令。其执行格式如下。

格式：

```
        JMP     REG
          或
        JMP     WORD  PTR  ADDR
```

其中，

（1）REG 表示的是通用寄存器，执行的操作为“(IP)←通用寄存器 REG 内容”。

（2）ADDR 表示的是各种寻址方式确定的存储单元地址，并且在无歧义的情况下，WORD　PTR 可以省略。执行的操作为“(IP)←字存储单元 ADDR 的内容”。

例如：

```
        JMP     AX
        JMP     WORD   PTR [BX]
        JMP     VARE                      ; VARE 为定义的变量
        JMP     VARE[SI]                  ; VARE 为定义的变量
        JMP     WORD   PTR [BP]
```

上例中第一条指令表示通用寄存器 CX 中存放有转移的目标地址。第二条指令表示目标地址存放在[BX]指向的一个字存储单元。第三条指令中 VARE 是由伪指令 DW 定义的字变量，指令转移地址在由 VARE 所指的字存储单元中，即指令转移地址不是 VARE，而是 VARE 单元的内容。这是变量和标号不同之处。也就是说，跳转指令中的标号表示转移地

址，而变量则是存放转移地址的地址。假设 DS=1000H，VARE=1286H，(1286H)=1224H，指令执行后，IP=1224H，即程序转移到代码段中偏移地址 1224H 处，而不是转移到变量 VARE（地址 1286H）处。

4. 段间直接转移

段间直接转移是指从这一代码段转移到另一个代码段中，因此，操作数是一个远标号，在另一个代码段内。其格式如下。

格式：

```
        JMP      FAR  PTR  TARGET
```

其中，

（1）TARGET 是目标标号，具有 FAR 属性，当无歧义时，也可用 JMP TARGET 代替 JMP　FAR　PTR　TARGET。

（2）FAR　PTR 表示段间转移的操作符。

段间直接转移执行的操作如下：

（1）(IP) ←TARGET 偏移地址。

（2）(CS) ←TARGET 的段地址。

例如：

```
    CODE1    SEGMENT
             …
    TARGET   LABEL    FAR
             …
    CODE1    ENDS
    CODE2    SEGMENT
             …
             JMP      FAR  PTR  TARGET
             …
    CODE2    ENDS
```

上例中“JMP　FAR　PTR　TARGET”指令也可写成“JMP　TARGET”。

段间直接转移的指令编码格式为 5 字节，其中，两个字节存放 JMP 指令 TARGET 的偏移量，两个字节存放 TARGET 的段基值。执行段间转移直接寻址指令时，就把它们分别送入 IP 与 CS，以实现段间转移。

5. 段间间接转移

段间间接转移与段间直接转移指令一样，也用于段间转移，只不过当前 CS 和 IP 由指令指明的存储器中连续的两个字更新，低位地址的字更新 IP，高位地址的字更新 CS，存放新 IP 和 CS 的存储单元地址由存储器操作数的寻址方式决定。其格式如下。

格式：

```
        JMP      DWORD  PTR  ADDR
```

其中，

ADDR 为存储器地址，是由存储器寻址方式确定的存储器地址，并且在无歧义的情况下，DWORD　PTR 可以省略。

指令的操作为：

（1）IP←ADDR 的第一个字存储单元中的内容。

（2）CS←ADDR 的第二个字存储单元中的内容。

例如：

```
JMP     DWORD  PTR  VAR [BX]          ; VAR 为变量
JMP     VAR_DWORD                     ; 省略了 DWORD PTR
JMP     DWORD  PTR  [BX][SI]
```

第一条指令表示目标地址存放在由 VAR [BX]指向的一个双字存储单元中，即取 DS 段中偏移为[VAR+BX]处的双字作为新的 CS 和 IP，第二条指令表示目标地址存放在由双字变量 VAR_DWORD 确定的一个双字存储单元中，即“IP←(VAR_DWORD，VAR_DWORD+1);CS←(VAR_DWORD+2，VAR_DWORD+3)”。第三条指令表示目标地址存放在由 [BX][SI] 确定的一个双字存储单元中，即“IP←([BX+SI]，[BX+SI]+1);CS←([BX+SI]+2，[BX+SI]+3)”。

7.3.2 条件转移指令

Intel 8086/8088 CPU 指令系统提供了 18 条条件转移指令，它们根据某标志位（如符号位（SF）、零值位（ZF）、进位位（CF）、奇偶位（PF）和溢出位（OF）等）或某些标志位的逻辑运算来判别条件是否成立。如果条件成立，则转移；否则继续顺序执行。操作码构成方法是，第一个字母是 J，紧接着的字母是标志寄存器的代号 P、S、Z、…，表示“条件”。其格式如下。

格式：

```
JXX     TARGET
```

其中，

（1）XX 为特定的 1 个或 2 个字母，表示相应的条件。

（2）TARGET 一般为标号。

执行的操作为“IP←IP+TARGET 的偏移量”。

例如：

```
JZ
JNZ
```

上例中第一条指令表示结果为 0，则转移。第二条指令表示结果不为 0，则转移。有一个条件转移指令 JCXZ，构成方法较特殊，它的转移条件是寄存器 CX 的内容为 0，而不是标志寄存器的某位标志为 0。

条件转移指令只能在段内进行直接寻址，并且是在从下一条指令的地址算起-128～127 个字节的地址范围内转移，而且指令执行时并不影响标志寄存器的各位。

条件转移指令一般用在算术运算、逻辑运算或移位运算等对标志寄存器有影响的指令之后，根据这些指令操作后影响的状态标志决定是否发生转移。

现将所有的条件转移指令列举如下。

1. 根据单个标志的设置情况进行转移

根据单个标志位的状态判断转移条件，该转移指令共有 5 种（10 条），其转移指令如表 7.1 所示。

指令中除了 C、Z、S、O 与 P 表示相应指令的 CF、ZF、SF、OF 与 PF 标志位的首字母外，其中 N 是英文 Not（否）的首字母，JE/JNE 中的 E 是英文 Equal（相等）的首字母，

JPE 中的 E 是英文 Even（偶数）的首字母，JPO 中的 O 是英文 Odd（奇数）的首字母。

表 7.1　简单条件转移指令

指　令	标 志 位	转移条件	含　义
JC	CF	CF=1	有进位或借位则转移
JNC		CF=0	无进位或借位则转移
JE/JZ	ZF	ZF=1	相等或等于 0 则转移
JNE/JNZ		ZF=0	不相等或不等于 0 则转移
JS	SF	SF=1	是负数则转移
JNS		SF=0	是正数则转移
JO	OF	OF=1	有溢出则转移
JNO		OF=0	无溢出则转移
JP/JPE	PF	PF=1	有偶数个 1 则转移
JNP/JPO		PF=0	有奇数个 1 则转移

注：表中 CF、ZF、SF、OF 和 PF 分别表示状态标志的进位位、零值位、符号位、溢出位和奇偶位。

【例 7.5】 写出程序片段，测试 AX 的低四位是否全是 0，如果均是 0，那么使 CX=0；否则使 CX=−1。

代码如下：

```
        …
        MOV     CX,-1
        TEST    AX,0FH          ; 测试 AX 的低 4 位
        JNZ     NEXT            ; 不全为 0 则转移到 NEXT
        MOV     CX,0            ; 全为 0 时使 CX=0
NEXT:   …
```

说明：

例中“JNZ　NEXT”指令是根据零值位 ZF 来决定是否执行转移的。如果 ZF=0，表示上一步测试 AX 的低 4 位不为 0，则转移到 NEXT；如果 ZF=1，表示上一步测试 AX 的低 4 位为 0，则不转移。

2. 比较两个无符号数，并根据比较结果的高低转移

比较两个无符号数，并根据比较结果的高低转移，假设在条件转移指令前使用比较指令，比较两个无符号数 DATA1 和 DATA2，指令进行的操作是 DATA1−DATA2，其转移指令如表 7.2 所示。

表 7.2　无符号数条件转移指令

指　令	转移条件	含　义
JA/JNBE	CF=0 并且 ZF=0	表示 DATA1>DATA2 转移
JAE/JNB	CF=0 或者 ZF=1/CF=0	表示 DATA1≥DATA2 转移
JB/JNAE/JC	CF=1 并且 ZF=0/CF=1	表示 DATA1<DATA2 转移
JBE/JNA	CF=1 或者 ZF=1	表示 DATA1≤DATA2 转移

注：（1）表中指令中 A 和 B 分别是英文 Above 与 Below 的首字母，N 是英文 Not 的首字母，E 是英文 Equal 的首字母。

（2）对于两个无符号数相减，是否产生借位和结果是否为 0 这两个条件是判断两数大小、相等的关键，所以无符号数条件转移指令使用进位位 CF 和零值位 ZF 来判断。

下面对这些指令进行详细介绍，并假设转移指令之前进行了两个数“DATA1”和“DATA2”的比较，无符号数条件转移指令如下。

（1）JA（或 JNBE）指令

JA（或 JNBE）指令是当 DATA1>DATA2 时，则转移；否则不转移。其格式如下。

格式：

```
JA      TARGET
   或
JNBE    TARGET
```

其中，TARGET 表示标号。

测试条件：CF=0（DATA1−DATA2 不产生借位）且 ZF =0（DATA1−DATA2 不为 0）。

（2）JNA（或 JBE）指令

JNA（或 JBE）指令是当 DATA1≤DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
JNA     TARGET
   或
JBE     TARGET
```

其中，TARGET 表示标号。

测试条件：CF=1（DATA1−DATA2 产生借位）或 ZF =1（DATA1−DATA2 为 0）。

（3）JB（或 JNAE,JC）指令

JB（或 JNAE,JC）指令是当 DATA1<DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
JB      TARGET
   或
JNAE    TARGET
   或
JC      TARGET
```

其中，TARGET 表示标号。

测试条件：CF=1（DATA1−DATA2 产生借位）。

（4）JNB（或 JAE,JNC）指令

JNB（或 JAE,JNC）指令是当 DATA1≥DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
JNB     TARGET
   或
JAE     TARGET
   或
JNC     TARGET
```

其中，TARGET 表示标号。

测试条件：CF=0（DATA1−DATA2 没产生借位）。

【例 7.6】 写出程序片段，实现两个无符号数（设在 AX 和 BX 中）的比较，把较大的数存放在 AX 中，把较小的数存放在 BX 中。

代码如下：

```
        …
        CMP     AX,BX           ; AX 与 BX 比较
        JAE     NEXT            ; 若 AX≥BX 则转移到 NEXT
        XCHG    AX,BX           ; AX 与 BX 交换
NEXT:   …
        …
```

3. 比较两个带符号数，并根据比较结果的大小转移

对于带符号数的比较，需要使用符号标志位 SF、溢出标志位 OF 和零标志位 ZF 来判断。假设在条件转移指令前使用比较指令，比较两个无符号数 DATA1 和 DATA2，指令进行的操作是 DATA1−DATA2，其转移指令如表 7.3 所示。

表 7.3　有符号数条件转移指令

指　令	转移条件	含　义
JG/JNLE	SF=OF 并且 ZF=0	表示 DATA1>DATA2 转移
JGE/JNL	SF=OF 或者者 ZF=1	表示 DATA1≥DATA2 转移
JL/JNGE	SF≠OF 并且 ZF=0	表示 DATA1<DATA2 转移
JLE/JNG	SF≠OF 或者 ZF=1	表示 DATA1≤DATA2 转移

注：指令中 G 和 L 分别是英文 Greater 与 Less 的首字母，N 是英文 Not 的首字母，E 是英文 Equal 的首字母。

下面分别分析各指令的转移条件（并假设之前进行了 DATA1−DATA2 的操作）。

（1）JNLE（或 JG）指令

JNLE（或 JG）指令是当 DATA1>DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
        JNLE    TARGET
          或
        JG      TARGET
```

其中，TARGET 表示标号。

测试条件：(SF ⊕ OF=0)并且 ZF=0。

（2）JNL（或 JGE）指令

JNL（或 JGE）指令是当 DATA1≥DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
        JNL     TARGET
          或
        JGE     TARGET
```

其中，TARGET 表示标号。

测试条件：SF ⊕ OF=0。

（3）JL（或 JNGE）指令

JL（或 JNGE）指令是当 DATA1<DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
        JL  TARGET
          或
```

```
        JNGE    TARGET
```

其中，TARGET 表示标号。

测试条件与功能：SF ⊕ OF=1。

（4）JLE（或 JNG）指令

JLE（或 JNG）指令是当 DATA1≤DATA2 时，则转移；否则不转移。其指令格式如下。

格式：

```
        JLE     TARGET
          或
        JNG     TARGET
```

其中，TARGET 表示标号。

测试条件与功能：SF ⊕ OF=1 或 ZF =1。

【例 7.7】 写出程序片段，实现两个有符号数（设在 AX 和 BX 中）的比较，把较大的数存放在 AX 中，把较小的数存放在 BX 中。

程序段如下：

```
            …
            CMP     AX,BX           ; AX 与 BX 比较
            JGE     NEXT            ; 若 AX≥BX 则转移到 NEXT
            XCHG    AX,BX           ; AX 与 BX 交换
NEXT:       …
            …
```

4. 测试 CX 的值为 0，则转移

测试 CX 的值为 0，则转移；否则不转移。其指令格式如下。

格式：

```
        JCXZ    TARGET
```

其中，TARGET 表示标号。

测试条件：（CX）= 0。

7.3.3 单分支程序设计

单分支类程序处理的逻辑比较简单，单分支类程序适合这种情况：当程序的逻辑根据某一条件为真时，程序转去执行目标代码，当程序逻辑为假时，程序的处理为空。

【例 7.8】 编写程序比较两个 8 位数大小，并把较大的数存入字节变量 MAX 中。

（1）分析问题

假设两个 8 位数存放在变量 VARW1 和 VARW2 中，确定分支的条件是判断变量 VARW1 和 VARW2 的大小。

（2）确定算法

采用比较转移指令，在汇编语言中用 AL 来保存中间结果。这是因为读取寄存器比读取存储器要快，而且汇编语言指令不允许两个操作数都为存储单元。

（3）画程序流程图

流程图如图 7.9 所示。

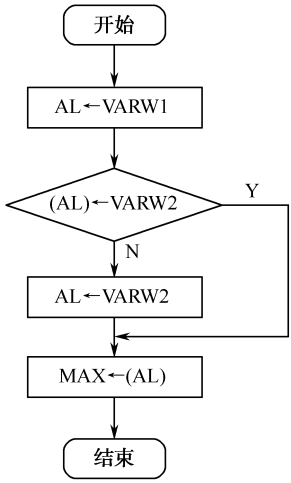


图 7.9　流程图

（4）分配存储空间和工作单元

该汇编语言程序的基本框架至少要两个段：数据段和代码段。数据段中三个数 VARW1、VARW2 和 MAX 都是 8 位数，应选 DB 类型。

（5）编写程序

程序如下：

```
DATA    SEGMENT                 ; 定义数据段
VARW1   DB   0004H              ; 定义字变量 VARW1
VARW2   DB   0008H              ; 定义字变量 VARW2
MAX     DB   ?                  ; 定义字节变量
DATA    ENDS                    ; 数据段定义结束
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:
        MOV     AX,DATA
        MOV     DS,AX
        MOV     AL,VARW1
        CMP     AL,VARW2        ; 比较 VARW1 和 VARW2
        JA      EXIT            ; 若 VARW1>VARW2，跳转到 EXIT；否则执行下列语句
        MOV     AL,VARW2        ; 将 VARW2 送到寄存器 AL 中
EXIT:   MOV     MAX,AL          ; 将(AL)送到存储单元 MAX 中
        MOV     AH,4CH
        INT     21H             ; 返回 DOS
CODE    ENDS                    ; 代码段结束
        END     START           ; 程序结束
```

7.3.4　双分支程序设计

双分支程序是比较常用的程序设计模式，它适合这种情况：当程序的逻辑根据某一条

件为真或假时，程序分别转去不同的程序处理段。

使用双分支程序设计模式时需要注意，为了不让“程序段 1”执行完成后进入到“程序段 2”中，在“程序段 1”的最后应该加一条无条件转移指令，跳转到分支类程序的出口。

【例 7.9】 编写一个程序，实现把一位十六进制数转换为对应的 ASCII 码。

（1）分析问题

十六进制数码与对应 ASCII 码的关系如下所示：

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30H	31H	32H	33H	34H	35H	36H	37H	38H	39H	41H	42H	43H	44H	45H	46H

假设十六进制数为 X，ASCII 码为 Y，则这种对应关系可表示为一个分段函数：

$$Y=\begin{cases}X+30H(0\leqslant X\leqslant 9)\\ X+37H(0AH\leqslant X\leqslant 0FH)\end{cases}$$

所以程序要根据十六进制数码值是否超过 9 而进行分支。

（2）确定算法

采用比较转移指令，在汇编语言中用 AL 来保存中间结果。

（3）画程序流程图

流程图如图 7.10 所示。

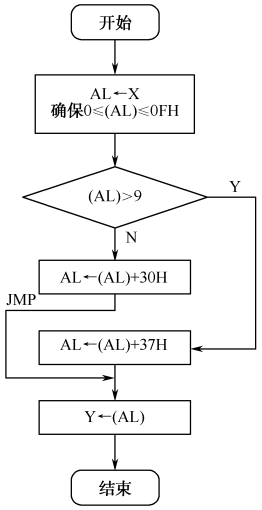


图 7.10 流程图

（4）分配存储空间和工作单元

该汇编语言程序的基本框架至少要两个段：数据段和代码段。数据段中两个数 X 和 Y，都是 DB 类型，X 存放要查询的 16 进制数据，Y 存放对应的 ASCII 码。另外，还要用到寄存器 AL。

（5）编写程序

程序如下：

```
DATA    SEGMENT                    ; 定义数据段
        X       DB   8             ; 假设要查找的十六进制数码
        Y       DB   ?             ; 存放对应的 ASCII 码
DATA    ENDS                       ; 数据段结束
CODE    SEGMENT                    ; 定义代码段
        ASSUME  CS: CODE,DS: DATA
START:  MOV     AX,DATA
        MOV     DS ,AX
        MOV     AL,X               ; X 存放在 AL 中
        AND     AL,0FH             ; 确保在 0 至 F 之间
        CMP     AL,9               ; AL 与 9 比较
        JA      NEXT1              ; 超过 9 转移
        ADD     AL,30H             ; AL 与 30H 相加并存放在 AL 中
        JMP     NEXT2              ; 跳转到 NEXT2
```

```
NEXT1:  ADD     AL,37H          ; AL 与 37H 相加并存放在 AL 中
NEXT2:  MOV     Y,AL            ; 加 AL 存放在 Y 中
        MOV     AH,4CH
        INT     21H             ; 返回 DOS 系统
CODE    ENDS                    ; 代码段结束
        END     START           ; 程序结束
```

说明：

例 7.9 中“JMP　NEXT2”转移语句（指令）很重要，如果没有这条指令，分支的结构就简化成为单分支结构。

由于上述程序分支的一边稍作变形后，可包含分支的另一边，所以进行代码转换工作可作如下优化，从而使处理既简单又高效。

```
        …
        ADD     AL, 30H
        CMP     AL,39H
        JBE     NEXT2
        ADD     AL,7
NEXT2 : …
```

一般情况下，如果是双分支且有一边很简单时，可考虑把它改变为单分支结构。具体的方法是，在判断之前先假设是简单的情况。

7.3.5　多分支程序设计

多分支结构程序的设计比前两种分支要复杂些。关键是怎样根据条件对多分支进行判断，确定不同分支程序转移的入口地址。这里列出了以下几种方法。

1. 逻辑分解流程图法

根据逻辑分解流程图，按照判别条件的先后，逐个进行判断和转移。逻辑分解流程图如图 7.7 所示。

【例 7.10】　编写一个程序，计算下式表示的函数值。其中，变量 X 和 Y 是有符号字变量。

$$Y=\begin{cases} X+10 & (-128\leqslant X<0) \\ 30X & (0\leqslant X\leqslant 10) \\ X-9 & (10<X\leqslant 127) \end{cases}$$

（1）分析问题

这是多分支结构。本题有 3 个分支：−128 ⩽X< 0、0 ⩽X⩽10 和 10 <X⩽127。按照逻辑分解的方法，可以先将其归并为两个条件：X⩾0 和 X< 0，由此形成两个分支；再将分支 X⩾0 分解为 X⩽10 和 10 <X⩽127，各分支均用条件转移指令来实现。

（2）确定算法

采用比较转移指令。

（3）画程序流程图

流程图如图 7.11 所示。

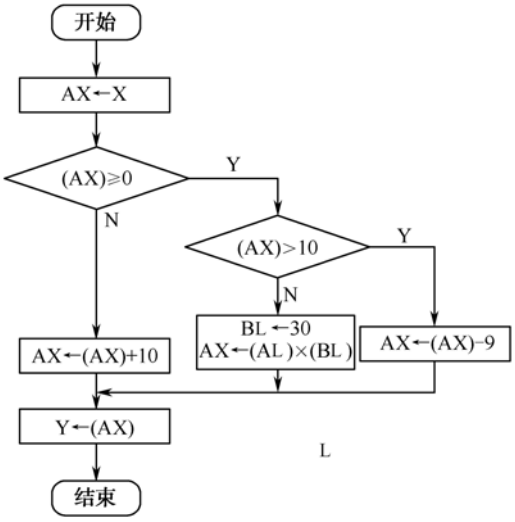


图 7.11　逻辑分解流程图

（4）分配存储空间和工作单元

该汇编语言程序的基本框架至少要两个段：数据段和代码段。数据段中至少定义两个变量：X 为 8 位数，应选 DB 类型。考虑到 y 的值可能会超过 8 位，所以选 DW 类型。假设任意给定的值存放在 X 单元中，函数计算结果的值存放在 Y 单元中。

（5）编写程序

程序如下：

```
DATA    SEGMENT                          ; 定义数据段
        X    DB   11D                    ; 假设要计算的十进制数
        Y    DW   ?                      ; 存放计算的结果
DATA    ENDS                             ; 数据段结束
CODE    SEGMENT                          ; 代码段开始
        ASSUME  CS: CODE,DS: DATA
START:  MOV     AX, DATA
        MOV     DS, AX
        SUB     Ah, AH
        MOV     AL, X                    ; 将 X 存放在 AX 中
        CMP     AX, 0                    ; 与 0 进行比较
        JGE     NEXT1                    ; 如果≥0 则跳转到 NEXT1
        ADD     AX, 10                   ; 第一种情况的计算结果
        JMP     RESULT                   ; 跳转到 RESULT
NEXT1:  CMP     AX, 10D                  ; 与 10 进行比较
        JG      NEXT2                    ; 如果>10 则跳转到 NEXT2
        MOV     BL, 30D
        IMUL    BL                       ; 第二种情况的计算结果
        JMP     RESULT                   ; 跳转到 RESULT
NEXT2:  SUB     AX, 9                    ; 第三种情况的计算结果
RESULT: MOV     Y, AX                    ; 把计算结果保存到变量 Y 中
        MOV     AH, 4CH
```

```
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS                        ; 代码段结束
        END     START               ; 程序结束
```

说明：

例中 X 的取值范围是在-128～127 之间。

对于多分支程序，如果采用逻辑分解流程图法实现，则 N 条分支需要（N-1）个条件转移指令完成。采用这样的方法编制的程序不够清晰，而且程序转移的范围可能超过 128 字节。在这个范围，采用条件转移指令实现起来不方便。跳转表是实现多分支结构程序的常用办法，跳转表可分为“跳转地址表法”和“跳转指令表法”。

2. 跳转地址表法

如果各分支的判定结果可以用一个寄存器不同的有序的值来表示，则可将各分支程序的入口地址与寄存器不同的有序的值一一对应，列入一张表中，此表称为“跳转地址表”。跳转表是由入口地址构成的数据表，该表可在数据段中定义，也可在代码段中定义，用来存放各分支程序段的入口地址。在程序执行时，根据所形成的值在表中查找到对应的入口地址，然后利用无条件转移指令跳转到相应的分支程序入口处继续执行。

例如，某个程序需 N 路分支，每路程序的入口地址分别为 BRANCH1、BRANCH2、…、BRANCHN。把这些转移的入口地址组成一个表，如图 7.12 所示。表内每两个字节存放一个入口地址的偏移量。若涉及跨段的转移，则入口地址需要 4 个字节存放，其中低位字存放入口处的偏移地址，高位字存放入口处的段地址。当根据条件得到分支号后，可以计算出该分支程序的入口地址所在位置，取出该入口地址，就可以控制转移到欲调用的分支程序中执行。

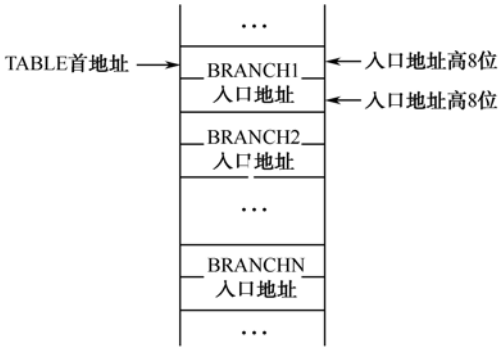


图 7.12　跳转地址表

段内各分支程序的入口地址的计算公式如下。

公式：

ARRAY=(N−1)×2+TABLE

其中，

（1）ARRAY 为各分支程序的入口地址。

（2）N 为分支号，可以存放在寄存器中。

（3）TABEL 为跳转地址表首地址。

这种利用跳转地址表实现多分支程序的转移的方法，称为“跳转地址表法”。

【例 7.11】 使用跳转地址表法实现如下 C 语言的 switch 语句，其中，变量 A 和 B 是有符号的整型（int）变量。

```
switch(A%8)
{
    case 0:
        B = 30;
        break;
    case 1:
    case 2:
        B = A + 22;
        break;
    case 3:
        B = 2*A;
        break;
    case 4:
        B--;
        break;
    case 5:
    case 6:
    case 7:
        printf("Function 5_6_7");
        break;
}
```

（1）分析问题

由题意可知，要求根据某种条件实现 5 个分支程序。其具体转移情况如下：

如果 A%8 为 0，则转至 CASE0。

如果 A%8 为 1 或 2，则转至 CASE12。

如果 A%8 为 3，则转至 CASE3。

如果 A%8 为 4，则转至 CASE4。

如果 A%8 为 5、6 或 7，则转至 CASE567。

（2）确定算法

采用跳转地址表法，结合条件转移指令来实现。可以把 5 个子程序的入口地址编成如图 7.12 所示的分支程序入口地址表。

（3）画程序流程图

流程图如图 7.13 所示。

（4）确定程序的基本框架

该汇编语言程序的基本框架要两个段：数据段和代码段。数据段中创建分支程序入口地址表，应选 DW 类型，使用寄存器 BX 定位。

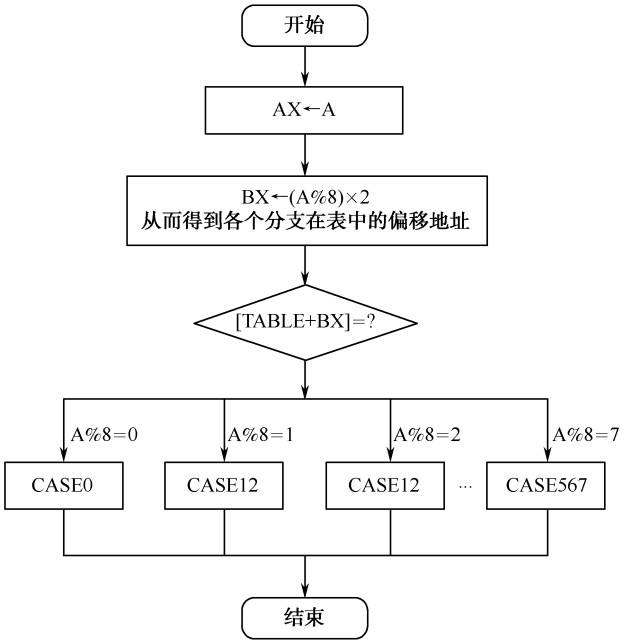


图 7.13　跳转指令表法流程图

（5）编写程序

程序如下：

```
DATA    SEGMENT
A       DW      78D
B       DW      ?                                   ; 存放计算结果
TABLE   DW      CASE0, CASE12, CASE12, CASE3        ;TABLE 即为地址表，用于存放各个
        DW      CASE4, CASE567, CASE567, CASE567    ; 处理程序入口标号处的偏移地址
MSG     DB      'Function 5_6_7$'
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS: CODE,DS: DATA
START:  MOV     AX, DATA                            ; 数据置代码段内
        MOV     DS, AX
        MOV     AX, A
        MOV     BX, AX
        AND     BX, 7                               ; 得到 BX 的低 3 位，实现 A%8 的计算
        SHL     BX, 1                               ; 由于地址表是字类型，其下标要乘 2
        JMP     TABLE[BX]                           ;（段内）间接转移 IP←[ TABLE+BX]
CASE0:  MOV     B, 30D                              ; 处理 A%8 为 0 的程序段
        JMP     NEXT
CASE12: ADD     AX, 22D                             ; 处理 A%8 为 1 或 2 的程序段
        MOV     B, AX
        JMP     NEXT
CASE3:  SHL     AX, 1                               ; 处理 A%8 为 3 的程序段
```

```
        MOV     B, AX
        JMP     NEXT
CASE4:  DEC     B                   ; 处理 A%8 为 4 的程序段
        JMP     NEXT
CASE567: LEA    DX, MSG             ; 处理 A%8 为 7.、8、9 的程序段
        MOV     AH, 9
        INT     21H                 ; 显示 MSG 信息
NEXT:   MOV     AH, 4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START               ; 程序结束
```

说明：

用跳转地址表实现多路转移的关键在于，转移入口的地址表和转移情况可整数化。如果这两个要求有一个不满足，或很难构造，则无法使用该方法。例中 A%8（A 除以 8 的余数）其实是分支 N−1 的值。

3. 跳转指令表法

跳转指令表法的思路与跳转地址表法基本相同，不同之处在于，它是在代码段中把转移到各个分支程序段的跳转指令存放在一个表中，该表就称为“跳转指令表”。

例如：

```
BRANCH:
        JMP   SHORT   BRANCH0
        JMP   SHORT   BRANCH1
        JMP   SHORT   BRANCH2
        JMP   SHORT   BRANCH3
```

上例中 BRANCH0、BRANCH1、BRANCH2 和 BRANCH3 为各个分支的入口地址（标号）。这样，通过执行跳转指令表中相应的跳转指令，就可以转移到不同的分支。由于短跳转指令的机器代码为两个字节，因此，可以将从转移条件得到的编号（从 0 开始）代入下面的公式，来计算出表内跳转转移指令的地址。

公式：

ARRAY=(N-1)×2+TABLE

其中，

（1）ARRAY 为表内跳转指令表的地址。

（2）N 为编号。

（3）TABEL 跳转指令表的首地址。

这种利用跳转地址表实现多分支程序的转移的方法，称为“跳转指令表法”。

【例 7.12】 使用跳转指令表改写【例 7.11】程序。

删除【例 7.11】数据段中的跳转地址表，在代码段中构建跳转指令表，通过执行跳转指令表中相应的跳转指令，就可以转移到不同的分支。流程图如图 7.14 所示。

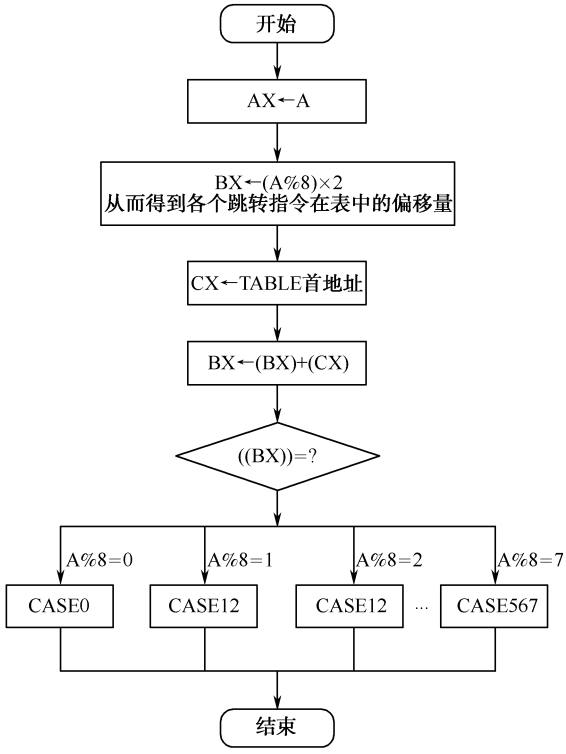


图 7.14　跳转指令表法流程图

程序如下：

```
DATA    SEGMENT
A       DW      78D
B       DW      ?                       ; 存放计算结果
MSG     DB      'Function 5_6_7$'
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS: CODE,DS: DATA
START:  MOV     AX, DATA                ; 数据置代码段内
        MOV     DS, AX
        MOV     AX, A
        MOV     BX, AX
        AND     BX, 7                   ; 得到 BX 的低 3 位，实现 A%8 的计算
        SHL     BX, 1                   ; 由于地址表是字类型，其下标要乘 2
        MOV     CX, OFFSET  TABLE       ; 将转移指令跳转表的首地址送到 CX 中
        ADD     BX, CX                  ; BX 中即为要跳转的地址
        JMP     BX                      ; 跳转到 BX 指示的跳转指令地址
TABLE:                                  ; TABLE 即为跳转表
        JMP     SHORT   CASE0
        JMP     SHORT   CASE12
        JMP     SHORT   CASE3
        JMP     SHORT   CASE4
```

```
        JMP     SHORT   CASE567
CASE0:  MOV     B, 30D              ; 处理 A%8 为 0 的程序段
        JMP     NEXT
CASE12: ADD     AX, 22D             ; 处理 A%8 为 1 或 2 的程序段
        MOV     B, AX
        JMP     NEXT
CASE3:  SHL     AX, 1               ; 处理 A%8 为 3 的程序段
        MOV     B, AX
        JMP     NEXT
CASE4:  DEC     B                   ; 处理 A%8 为 4 的程序段
        JMP     NEXT
CASE567: LEA    DX, MSG             ; 处理 A%8 为 7.、8、9 的程序段
        MOV     AH, 9
        INT     21H                 ; 显示 MSG 信息
        JMP     NEXT
NEXT:   MOV     AH, 4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START               ; 程序结束
```

说明：

跳转指令表中的每一项就是一条无条件转移指令，而跳转指令表是代码段中的一段程序。例中 A%8（A 除以 8 的余数）其实是编号 N−1 的值。

7.4 循环结构程序设计

在实际应用中，经常需要重复执行一段相同的程序，这需要循环程序来实现。转移指令也可以用来设计循环程序。在 Intel 8086/8088 指令系统中，为了简化循环程序的设计，使用了 4 条循环控制指令，以增强循环控制的灵活性。本节先介绍循环控制指令，然后讨论循环程序的设计方法。

7.4.1 循环控制指令

Intel 8086/8088 指令系统中提供了 4 条循环控制指令，分别是 LOOP、LOOPE/LOOPZ、LOOPNZ/LOOPNE 和 JCXZ。需要注意的是，这 4 条指令都隐含使用 CX 作为循环计数器。所以在指令中虽然没有出现 CX，但是在程序初始化时，一定要注意设置 CX 的初始值，也就是循环的次数。并且 LOOPE/LOOPZ 和 LOOPNZ/LOOPNE 指令还使用了标志位 ZF 作为循环控制条件。这 4 条指令均不影响标志位。都是采用段内直接寻址方式，操作数一般为标号。

1. LOOP

LOOP 指令首先使寄存器 CX 减 1，然后判断 CX 是否为 0，如果 CX 不为 0，则转移到由标号指定处执行；如果 CX 为 0，则顺序执行 LOOP 指令之后的指令。其格式如下。

格式：

```
LOOP    TARGET
```

其中，TARGET 表示的是标号。

测试条件：CX≠0

LOOP 指令的功能相当于高级语言中的 FOR 语句，在使用 LOOP 指令控制循环时，首先将循环的次数送 CX 寄存器，在循环体中，一遇到 LOOP 指令，CX 内容减 1，然后判断 CX 是否为 0，如果为 0，则退出循环；否则，继续执行。

2. LOOPZ/LOOPE

LOOPZ 和 LOOPE 是同一条指令的两个不同的助记符。该指令首先使寄存器 CX 减 1，然后判断，若 CX 不为 0，而且 ZF=1，则转移到由标号指定处执行；否则，则顺序执行 LOOPZ/LOOPE 指令后的指令。其指令格式如下：

格式：

```
LOOPZ   TARGET
   或
LOOPE   TARGET
```

其中，TARGET 表示的是标号。

测试条件：ZF=1 且 CX≠0

3. LOOPNZ/LOOPNE

LOOPNZ 和 LOOPNE 是同一条指令的两个不同的助记符。该指令首先使寄存器 CX 减 1，然后判断，若 CX 不为 0，而且 ZF=0，转移到由标号指定处执行；否则，则顺序执行下一条指令。

格式：

```
LOOPNZ      TARGET
   或
LOOPNE      TARGET
```

其中，TARGET 表示的是标号。

测试条件：ZF=0 且 CX≠0

4. JCXZ

JCXZ 指令首先检测 CX 是否为 0，如果寄存器 CX 等于 0 时，转移到由标号指定处执行；否则，则顺序执行。

格式：

```
JCXZ    TARGET
```

其中，TARGET 表示的是标号。

JCXZ 并不对 CX 寄存器进行修改。测试条件：CX=0

注意：

上述循环指令的执行并不影响标志位。

【例 7.13】 设有一个首地址为 ARRAY 的 LENGTH 个字的数组，试编写一个程序段，求出该数组的各元素之和（不考虑溢出），并把结果存入变量 TOTAL 中。

程序片段如下：

```
        …
        MOV     CX, LENGTH          ; 设置循环计数器初始值
        MOV     AX, 0               ; 结果置初值

        MOV     SI, 0               ; 偏移量 SI 置初值
LOOP1:  ADD     AX, ARRAY[ SI]      ; 当前数组元素加入结果中
        ADD     SI, 2               ; 修改数组元素表指针指向下一个字元素
        LOOP    LOOP1               ; 判断数组元素是否累加完毕？若未完，继续
        MOV     TOTAL，AX           ; 数组元素累加结束，将结果送人 TOTAL 中
        …
```

7.4.2 循环程序的结构

一个循环通常由设置循环变量初值、循环语句序列、修改循环变量值、控制和结束处理五个部分组成。

1. 设置循环变量初值部分

循环程序都要求定义一个变量，其值的变化决定继续循环或终止循环，称为循环变量。在进入循环之前，都要通过程序或变量定义语句给出该变量的值。

2. 循环语句序列部分

循环语句序列是循环程序的工作部分需要重复执行的程序段，完成循环程序欲达到的功能，可由顺序程序、分支程序组成，也可由另一循环程序担当。若只有顺序程序或分支程序，这样的循环程序称为单重循环程序；若是另一个循环程序，则叫多重循环程序。

3. 修改循环变量值部分

当循环有限次数后，循环程序应当结束循环，继续执行循环体后面的程序，否则将会形成死循环。这就要求，在每次循环时（在循环体内）按规律修改循环变量的值，而且应当是向能使循环结束的方向改变，从而使循环能够正确地重复执行。通常把循环语句序列和修改循环变量值两部分合起来称为循环体。

4. 控制部分

判断循环结束条件是否成立。如满足循环条件，则进入新一轮循环过程，否则，退出循环体，开始后续操作。

循环程序的控制方法多种多样，一般由程序设计人员根据实际问题的需要，实现循环程序的控制。常用的循环控制方法一般有计数法与条件控制法两种。

计数法在循环次数已知的情况下比较方便。一般采用循环指令进行控制。循环指令能隐含修改循环计数器（CX 寄存器）的内容，并自动判断计数器状态，如计数器为 0，则中止循环。也可采用测试指令与条件转移指令一起进行计数控制。

条件控制法在循环次数未知的情况下，根据程序运行的状态来确定是否终止循环。一般采用测试指令和条件转移指令进行条件控制。

5. 结束处理部分

对循环结束进行适当处理，如存储结果等。有的循环程序可以没有这部分。

循环程序常见的结构形式有两种，如图 7.15 所示。

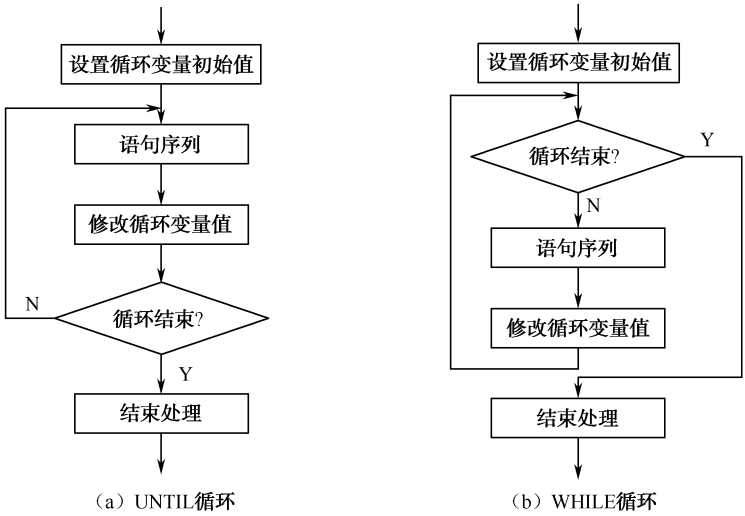


图 7.15　循环程序结构

UNTIL 循环如图 7.15（a）所示，其特点是先执行一次循环体内的语句，再判断是否满足循环条件，如果满足再进入循环体执行，直到条件满足时退出循环。如果进入时就不满足条件，也至少执行一次循环体内语句。

WHILE 循环如图 7.15（b）所示，其特点是进入循环后先判断是否满足循环条件，如果满足则进入循环体执行，每执行一次循环体内语句后重新判断条件表达式的值，直到条件不满足时退出循环。如果进入时就不满足条件，可能一次也不执行循环体内的语句。

7.4.3　单重循环程序设计

由顺序结构或分支结构组成循环体的循环结构程序称为单重循环程序（简称单循环程序）。单循环程序的循环体内不再包含循环结构。单循环程序设计比较简单，常用的循环控制方法有计数控制法、条件控制法和逻辑尺控制法等。下面分别加以讨论。

1．记数控制法

常见的循环是计数循环，当循环了一定次数后，就结束循环。在微型机中，常用一个内部寄存器（一般用 CX）或一对寄存器作为计数器，对它的初值置循环次数，每循环一次减 1，当计数器的值减为 0 时，就停止循环。也可以初值置为 0，每循环一次加 1，再与循环次数相比较，若两者相等，就停止循环。

【例 7.14】　以 ARRY 为首地址的数据段中存放着 N 个互不相等的 16 位有符号数，求出其中的最大数并存放到 MAX 字单元中。

（1）分析问题

可以任选一个数，作为比较的标准。比如，先把第一个数分别送入 AX，看做是最大数比较的标准，然后将数组后面的每一个数与其比较，如果它比 AX 中的当前值还大，那么就将该数存入 AX；否则不变。然后再和下个数进行比较。经 N-1 次循环比较后，AX 就是所要求的最大数。

（2）确定算法

采用比较循环指令，循环次数是 N-l 次。故用循环次数作为循环的控制条件，再配合使用 LOOP 指令。循环控制采用计数控制。

（3）画程序流程图

流程图如图 7.16 所示。

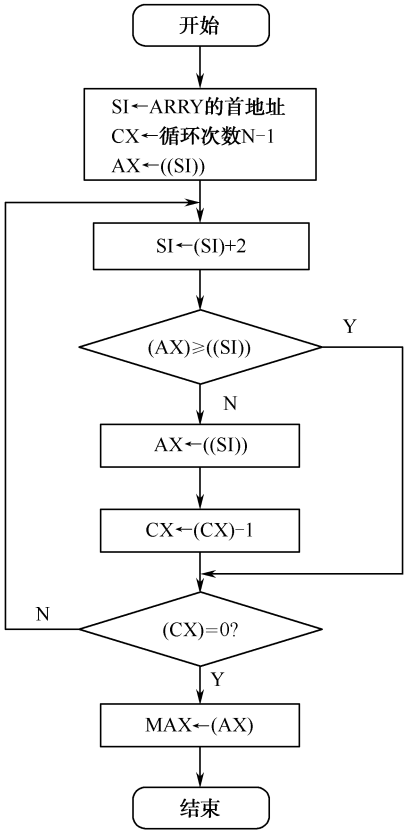


图 7.16　在数组中寻找最大值的流程图

（4）确定程序的基本框架

该汇编语言程序的基本框架至少需要两个段：数据段和代码段。数据段中至少定义 3 个变量：一个数组变量 ARRY，由题意可知应为 16 位数，应选 DW 类型，使用寄存器 SI 定位。最大数变量 MAX，由题意可知应为为 16 位数，选 DW 类型，其中间结果分别存放在寄存器 AX 中。还有一个数 N，为循环次数，计数器用 CX。

（5）编写程序

程序如下：

```
DATA    SEGMENT
ARRY    DW      248, -26, 693, 30, 742DH    ; 假设数组中有这些数据
        DW      8621H, 7766H, 8675H, 2986H, 5678
N       EQU     $-ARRY                      ; 计算数组个数
MAX     DW      ?                           ; 存放最大值
```

```
DATA    ENDS
CODE    SEGMENT
        ASSUME    CS:CODE, DS:DATA
START:  MOV     AX, DATA
        MOV     DS, AX
        LEA     SI, ARRY          ; 送 ARRY 的偏移地址到寄存器 SI
        MOV     CX, N-l           ; 送循环次数到寄存器 CX
        MOV     AX, [SI]          ; 取第一个数到 AX
LOP:    ADD     SI, 2             ; SI←SI+2，为取下一个数作准备
        CMP     AX, [SI]          ; AX 与下一个数比较
        JAE     AGAIN             ; 若 AX≥((SI))则转移到循环体 AGAIN
        MOV     AX, [SI]          ; 将比 AX 还大的数送入 AX 中
AGAIN:  LOOP    LOP               ; 若(CX)≠0 则转移到 LOP
        MOV     MAX, AX           ; 存入最大数
        MOV     AH, 4CH
        INT     21H               ; 返回 DOS 系统
CODE    ENDS
        END     START
```

说明：

从该程序中注意到，循环体内程序代码是相同的、不变的，但数据处理内容是不同的、变化的。循环程序设计的一大难题就是如何将看似不相同的内容使用相同的一组代码实现。例 7.14 中设计一个地址指针，在每次循环后改变该指针的值，实现操作数的改变。这是当处理方法相同时采用最多的方法。例 7.14 中的单循环为 UNTIL 循环，数组 ARRY 中至少要有两个元素，如果设计成 WHILE 循环，则不需要有此要求，请读者将其设计为 WHILE 循环。

2. 条件控制法

有些情况下循环次数事先无法确定，但可分析出循环结束的条件，然后利用这个条件来控制循环。控制循环的指令为条件转移指令。

【例 7.15】 写一个把字符串中的所有大写字母改为小写的程序。设字符串以 0 结尾。

（1）分析问题

由题意可知，题中的主体是一个循环次数不确定（字符串长不确定），需要根据是否到达字符串尾这个条件来控制的循环。

（2）确定算法

将字符串中各个字符和 0、'A'与'Z'进行比较，如果为 0，则退出程序，如果在'A'~'Z'，则转换为小写。

（3）画程序流程图

流程图如图 7.17 所示。

（4）确定程序的基本框架

该汇编语言程序的基本框架至少需要两个段：数据段和代码段。数据段中至少定义一个变量，一个字符串数组，应选 DB 类型，使用寄存器 SI 定位，其中间结果存放在寄存器 AL 中。

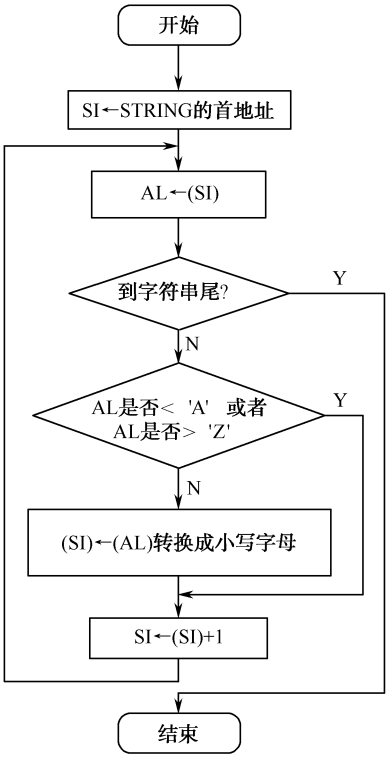


图 7.17　字符串中的字母小写化的流程图

（5）编写程序

程序如下：

```
DATE    SEGMENT
STRING  DB      'How aRE yoU   !', 0        ; 假设的字符串
DATE    ENDS
CODE    SEGMENT
        ASSUME  CS: CSEG, DS: DSEG
START:  MOV     AX,DATE
        MOV     DS, AX
        MOV     SI, OFFSET STRING          ; 取字符串始偏移地址送到 SI 中
AGAIN:  MOV     AL, [SI]                   ; 取一字符
        OR      AL, AL                     ; 判断是否到字符串尾?
        JZ      EXIT                       ; 若到字符串尾，转到 EXIT
        CMP     AL, 'A'                    ; 否则，判是否为大写字母
        JB      NEXT                       ; 不为大写字母则转到 NEXT
        CMP     AL, 'Z'
        JA      NEXT                       ; 不为大写字母则转到 NEXT
        OR      AL, 20H                    ; 是大写字母，改为小写字母
        MOV     [SI], AL                   ; 送回到字符串中
NEXT:   INC     SI                         ; 调整指针
        JMP     AGAIN                      ; 继续
EXIT:   MOV     AX, 4CH                    ; 结束，返回 DOS 系统
```

```
        INT     21H
CODE    ENDS
        END     START
```

3. 逻辑尺控制法

计数控制法和条件控制法是用来控制单一程序段重复执行次数的。但有些问题需要根据条件，分别控制两种不同的操作（或程序段）重复执行的次数。为了区别两种不同的操作，用二进制位的 1 表示执行第一种操作，用 0 表示执行第二种操作，并且在内存中设置一个字或字节单元，让它的各位分别记录各次操作的种类：是 1 类操作，还是 0 类操作。这个字或字节单元，称为“逻辑尺”。其中，1 和 0 的位数，分别代表要重复执行的两种操作的次数。进入循环后，通过对这个单元的顺序移位查找，识别标志位，就可以确定应该执行哪种操作。利用逻辑尺来进行循环控制的方法，称为“逻辑尺控制法”。

【例 7.16】 编写程序，求将元素个数为 16 的字节类型单维数组 X 与 Y 中对应元素相加或相减，结果（假设仍为字节类型）存放到数组 Z 中。即实现：X1+Y1→Z1，X2−Y2→Z2，…，X16+Y16→Z16（中间加或减没有规律）。

（1）分析问题

可以使用循环程序结构来完成这种问题。由题意可知，这里有两种操作：加法和减法。为了区别每次应该做哪一种操作，可以设立标志位，如标志位为 1，做加法；如标志位为 0，做减法。这样，进入循环后，只要判别标志位就可确定应该做的操作了。

（2）确定算法

采用循环指令，循环次数是 16 次。故用循环次数作为循环的控制条件，再配合使用 LOOP 指令。循环控制采用计数控制。先设置一单元 RULER，让其各位为 1 或 0，对应程序执行时每一次循环的加或减。假设运算次序是：加加加减加减加加减加加减减减减加，则 RULE 单元的内容为：1110101101100001B。

（3）画程序流程图

流程图如图 7.18 所示。

（4）汇编语言程序的基本框架

本框架至少需要两个段：数据段和代码段。数据段中至少定义 3 个变量：两个数组变量 X 和 Y、设为 8 位数，应选 DB 类型；和（差）变量 Z 设为 8 位数，应选 DB 类型；其中间结果存放在寄存器 DL 中；计数器用 CX，逻辑尺存放在寄存器 AX 中。

（5）编写程序

程序如下：

```
DATA    SEGMENT
RULER   DW  0EB61H                    ; 逻辑尺常数
X       DB  30H, 12H, 1BH, 2AH, 32H, 0B3H,34 , 23H, 28, 44H, 29, 50, 13, 25, 23, 27H
Y       DB  18H, 14H, 29, 50, 33, 25, 0A3H, 63, 25H, 22H, 2BH, 1AH, 52H, 0B3H,34
COUNT   EQU  ($-X)/2
Z       DB  16DUP (0)
DATA    ENDS
CODE    SEGMENT
        ASSUME    CS:CODE, DS:DATA
START:  MOV     AX, DATA
        MOV     DS, AX
```

```
        MOV     CX, COUNT           ; 数组元素个数
        MOV     AX, RULER           ; (AX)←逻辑尺常数
        MOV     SI, OFFSET    X     ; X 数组指针
        MOV     BX, OFFSET    Y     ; Y 数组指针
        MOV     DI, OFFSET    Z     ; 结果数组指针
LOOPl:  MOV     DL, [SI]            ; 取 X 一个数
        SHL     AX, 1               ; 取逻辑尺的一位到 CF 中
        JC      ADD1                ; (CF) =1，转到 ADD1
        SUB     DL, [BX]            ; 该次循环逻辑尺该位为 0，两数相减
        JMP     END1
ADDl:   ADD     DL, [BX]            ; 逻辑尺该次该位为 1，两数相加
ENDl:   MOV     [DI], DL            ; 结果送[DI]
        INC     SI                  ; 数组指针加 1
        INC     DI
        INC     BX
        LOOP    LOOP1               ; 循环控制
        MOV     AH, 4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START
```

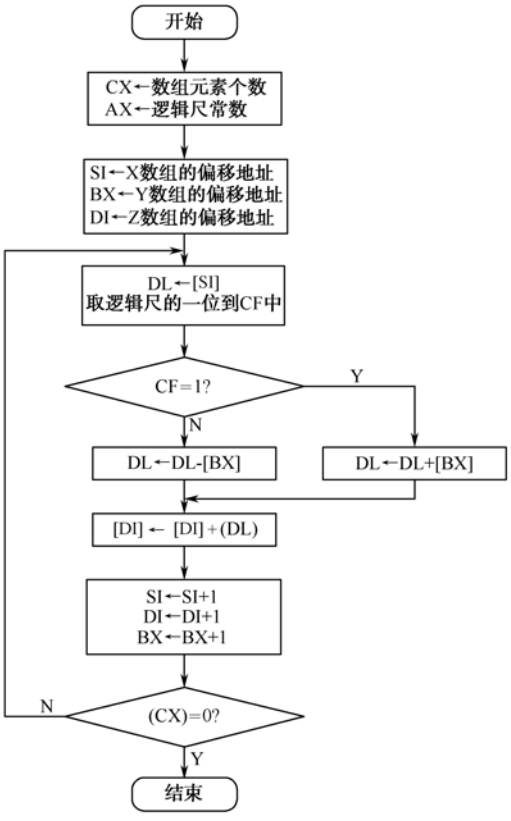


图 7.18　逻辑尺控制法流程图

说明：

注意该次序从左到右与处理的前后顺序相对应。如果处理元素不到 16 个，则后面的处理对应位可任意设置为 0 或 1。单元 RULER 相当于一把尺子，用做判别做加法运算还是减法运算，所以也将该尺子叫做逻辑尺，其十六进制代码（本例中即为 0EB61H）称之为逻辑尺常数。

本例中数组数据与逻辑尺常数都放在数据区，每次执行时要修改数据区中的数据。为方便使用，应当提供从键盘输入数组数据与逻辑尺常数的手段，这样用起来就更方便了。以上这种循环程序的设计方法，也叫逻辑尺控制法。它只是循环程序设计中一个典型的例子，在应对实际问题时，能利用循环程序实现的应当尽可能利用循环程序实现，这样不仅仅是节省程序量，减少内存的占用，而且使程序清晰，调试更方便，减少错误。只是需要在没有规律的情况下设计解决办法，使之有规律起来，这是需要开动脑筋并且不断进行经验积累的。

7.4.4　多重循环程序设计

如果一个循环程序的循环体内包含有一个或多个循环结构的程序，那么这个程序称为多重循环结构程序。多重循环既然是由单重循环嵌套而成的，所以多重循环和单重循环的设计方法是一致的，但应分别考虑各重循环的控制条件及其程序的实现，相互之间不要混淆。

在多重循环结构的设计中，主要应该掌握以下几点。

（1）应该完全包含在外循环的里面，成为外循环体的一个组成部分，不允许循环结构交叉。

（2）外层循环再次进入内层循环时，内层循环的初始条件必须重新设置。

（3）内循环的初值应该安排在进入内循环之前，但必须在外循环体内。

（4）循环中都使用寄存器 CX 作计数控制，那么由于只有一个计数寄存器 CX，因此需要利用堆栈保存与恢复 CX 寄存器的值。或者，在内层次上使用转移指令来实现循环的控制。

（5）只能从循环结构内转出或在同层循环内转移，而不能从另一个循环结构外转入该循环结构内。

【例 7.17】　编写一程序，求级数 $1^2+2^2+3^2+\cdots$ 的前 N 项的和。

（1）分析问题

本题程序可以采用双重循环。内循环计算级数各项的值，外循环计算各级数项之和。

（2）确定算法

对于 N^2 的计算采用连加的方法，即是：$N^2=N\times N=N+N+\cdots+N$，即 N 个 N 相加。此题可以在内循环中计算级数各项的值，即 N 个 N 相加，外循环计算各级数项之和。

（3）画程序流程图

流程图如图 7.19 所示。

（4）确定汇编语言程序的基本框架

本框架至少需要两个段：数据段和代码段。数据段中至少定义 2 个变量：存放计算结果变量应选 DW 类型；存放要计算级数的项数 N 应选 DB 类型。内循环计算结果存放在

AX 中，外循环计算结果存放在寄存器 DX 中。内循环计数器存放在 BX 中，外循环计数器用 CX。

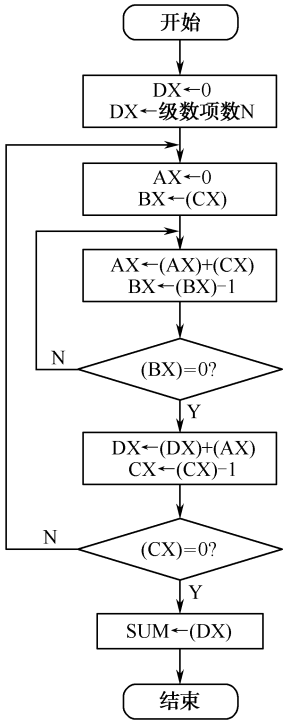


图 7.19　级数 $1^2+2^2+3^2+\cdots$的前 N 项和的流程图

（5）编写程序

程序如下：

```
DATA    SEGMENT
SUM     DW   ?                  ; 存放计算结果
N       DB   10                 ; 存放级数项数 N
DATA    ENDS
ODE     SEGMENT
        SSUME   CS:CODE, DS:DATA
TART:   MOV     AX, DATA
        MOV     DS, AX
        MOV     DX, 0           ; 存放级数的和
        MOV     CX, 0           ; CX 设置为 0
        MOV     CL, N           ; 设置外循环次数为 N
OP1:    MOV     AX, 0           ; 项的和赋初值 0
        MOV     BX, CX          ; 设置内循环次数为 CX
OP2:    ADD     AX, CX          ; 求级数项的值
        DEC     BX              ; 内循环减 1
        JNZ     LOP2            ; BX 计数不为 0，继续内循环
        ADD     DX,AX           ; 累加级数项
        LOOP    LOP1            ; CX 已在 LOOP 中减 1 操作，若不为 0，继续循环
```

```
        MOV     SUM, DX      ; 存级数和
        MOV     AH, 4CH
        INT     21H          ; 返回 DOS 系统
CODE    ENDS
        END     START        ; 程序结束
```

习题

1．程序的基本结构有哪三种？

2．条件转移指令之前的算术运算指令和逻辑运算指令通过何种方式为条件转移指令提供判断依据？

3．指出以下指定段的条件转移指令是怎样转移的。

（1）
```
XOR     AL, AL
JZ      LOP1
…
```

（2）
```
MOV     AL,6
CMP     AL, 3
JAE     LOP1
…
```

（3）
```
MOV     AL,2
SUB     BL,BL
MUL     BL
JNZ     LOP1
…
```

（4）
```
MOV     AL,-3
ADD     AL,3
JNZ     LOP1
…
```

（5）
```
MOV     AL, ‘D’
TEST    AL, 40H
JZ      LOP1
…
```

4．段内转移指令和段间转移指令操作过程的不同之处？

5．试比较 LOOPE 指令与 LOOPNZ 的异同点。

6．已知两个 8 位无符号数 X 和 Y，分别存放在 BUF 和 BUF+1 单元，且 X>Y。请编程计算 X−Y，结果存回 BUF 单元。

7．编写一个顺序结构程序，将 X 取补，设 X 为双字，X 的值及存放结果的单元自行定义。

8．在内存 BUFFER 单元中定义有 10 个 16 位数，试寻找其中的最大值及最小值，并存放在指定的存储单元 MAX 和 MIN 中。

9．在 BUFFER 开始的单元中存放着一个字符串，请判断该字符串中是否存在数字，

如果有则将 X 单元置 1；否则置 0。

10．在 DATA 单元有一个二进制数 X，要求编程完成运算：

$$Y=\begin{cases} X-1 & (-128\leqslant X<0) \\ X & (X=0) \\ X+1 & (0<X\leqslant 127) \end{cases}$$

11．在以 BUF 为首地址的字节存储区中存放有 n 个无符号数 X_1，X_2，X_3，…，Xn，现在要求将它们从小到大的顺序排列在 BUF 存储区中，试编写其程序。

12．已知从内存 DATA 单元开始连续存放若干个字符，编制程序求这些字符中出现次数最多的字符，将其出现次数及 ASCII 码分别存入 NUMB 和 DATAM 单元。

13．编写程序，已知 N，计算 1+3+5+7+…+(2×N−1)。

14．编写程序，输入一个数字 N，然后计算 N!。

15．编写程序，从键盘上输入一个数字 N，然后输出一个 N*N 的回形矩阵，当 N=5 时，回形矩阵的如下所示：

```
1   1   1   1   1
1   2   2   2   1
1   2   3   2   1
1   2   2   2   1
1   1   1   1   1
```

16．编写程序，首先在数据段中定义一个数据表，然后在表存放一组数字，存放顺序为升序。然后从键盘上输入任意一个数字，采用二分查找法判断输入的数字是否在表中。

二分查找表的主要思想是：首先把表中间的元素同要查找的元素进行比较，如果相同，则找到，如果要查找的元素小，则在前半个表查找，否则如果要查找的元素大，则在后半个表进行查找。

提示：首先在数据段中定义如下变量：

```
i    dw  ?                    ; 用于存放表首位置
j    dw  ?                    ; 用于存放表尾位置
m    dw  ?                    ; 用于存放表的中间位置
```

然后，根据查找结果不断更新上面三个变量。

17．编写程序，从键盘上输入一组数字，数字之间用空格分开，数据的输入以回车结束，然后采用选择排序法对数据进行排序，最后输出排序后的结果。

选择排序是这样一种排序方法，首先要在所有的数中找出最小者，将其存放到第一个位置上，然后在除第一个数外的所有数中寻找最小者，并将其存放到第二个位置，照此办理，直到只剩下一个数为止。

提示：在数据中定义如下变量：

```
n    dw  ?                    ; 要比较的数据的个数
i    dw  ?                    ; 最小元素的存放位置
j    dw  ?                    ; 开始比较的位置
k    dw  ?                    ; 临时最小值所在的位置
m    dw  ?                    ; 存放临时最小值
```

第 8 章

子程序设计

子程序是汇编语言程序设计的重要内容。子程序是一种解决重复性问题的重要设计方法，采用子程序结构可以简化源程序书写，提高程序存储效率，减少出错率，增加程序的易读性和可维护性，并且有利于子程序资源的组织和使用。本章着重介绍如何定义、调用与返回子程序，子程序设计的基本要求，以及如何在子程序与主程序间传递参数和子程序的嵌套与递归调用等内容。

8.1 子程序的定义

子程序是程序设计的基本概念。实际编程时，常把功能相对独立的程序段单独编写和调试，作为一个相对独立的模块供程序使用，这就是子程序，也称为过程，相当于高级语言中的过程和函数，调用子程序的程序称为主程序，或称为调用程序。

8.1.1 子程序的格式

在汇编语言中，要使某一程序段成为一个子程序，必须先通过过程定义伪指令 PROC 和 ENDP 来把它定义成为一个子程序。子程序（也称为“过程”）的定义格式如下：

格式：

```
PROC_NAME  PROC   [NEAR/FAR]     ; NEAR 表示过程在本代码段，FAR 表示在其他代码段
                  …              ; 过程体
                  RET            ; 返回指令
PROC_NAME  ENDP
```

其中，

（1）PROC_NAME 为子程序名（或称为过程名），它与标号类似，具有 3 种属性（偏移、段和类型）。它也是所定义子程序入口的符号地址。其段值和偏移量对应于子程序的入口地址，其类型就是该子程序的类型。类型属性中的 NEAR 和 FAR 指明该子程序被调用范围。NEAR 表示段内调用，即指调用程序（主程序）和子程序在同一个代码段中。如果一个子程序被定义为段内调用，那么在定义中可以省略 NEAR 不写。FAR 表示段间调用，即指调用程序和子程序分别在两个不同的代码段中。如果一个子程序被定义为段间调用，

那么在定义中不能省略 FAR。

（2）PROC、ENDP 是定义子程序时必须使用的保留字，PROC 和 ENDP 相当于一对括号，将子程序的指令包括在内。PROC 表示子程序的开始，ENDP 表示子程序的结束。

（3）RET 指令通常作为子程序的最后一条“指令”，用来控制 CPU 返回到主程序的调用处继续向下执行。

例如：

```
TRANASC     PROC    NEAR
            AND     AL,0FH
            ADD     AL,30H
            CMP     AL,39H
            JBE     TRTURN1
            ADD     AL,7
TRTURN1:    RET
TRANASC     ENDP
```

上例是一个段内调用名为“TRANASC”的子程序，它是把一位十六进制数转换为对应 ASCII 码的子程序，需转换的十六进制数码存放在 AL 的低 4 位。为了能返回调用程序，一般说来，在一个过程中至少要有一条返回指令，也可含多条返回指令。返回指令语句是过程的出口。当然返回指令不一定非要安排在过程的最后。

【例 8.1】 编写一个子程序 UPPER，实现把寄存器 AL 中存放的小写字母变成大写字母。

子程序如下：

```
UPPER   PROC    NEAR
        CMP     AL, 'a'             ;书写'a'的 ASCII 码 61H 也可以
        JB      OVER
        CMP     AL, 'z'
        JA      OVER
        SUB     AL,20H              ;书写指令 AND AL, 0DFH 也可以
OVER:   RET
UPPER   ENDP
```

说明：

因为此子程序被定义为段内调用，那么在定义中可以省略 NEAR 不写。

8.1.2 子程序的说明信息

编写子程序除了要考虑实现子程序功能的方法外，还要养成书写子程序说明信息的好习惯。子程序说明信息可以使用户无须查看子程序的内部结构或程序本身，就可正确地选择和使用子程序，通常会在子程序的程序代码前以注释的方式给出子程序的一些说明信息，主要用来说明子程序的功能、调用方法及相关参数。子程序的说明信息通常由以下几部分组成。

（1）子程序名称和调用类型。子程序名称应使用具有适当含义的标识符，调用类型指段内调用或段间调用。

（2）子程序功能描述。用自然语言或数学语言的形式简单清楚地描述子程序完成的任务。

（3）子程序的入口参数和出口参数。说明子程序要求有几个入口和出口参数，以及这些参数表示的意义及传递方式。

（4）受影响的寄存器。说明子程序运行后，哪些寄存器的内容会被破坏，以便使用者在调用该子程序之前注意保护寄存器信息。

（5）子程序的调用形式。当子程序的第一条语句不是子程序的入口指令时，必须给出调用形式，这时一般应以子程序入口指令处的标号或地址作为 CALL 指令的操作数。

（6）子程序的编写者。可选项，为将来的维护提供信息。

（7）子程序的编写日期。可选项，用于确定程序是否是最新版本。

这些说明性信息虽然不是子程序功能的一部分，但其他程序员可通过它们对该子程序的整体信息有一个较清晰的认识，为准确地调用它们提供直接的帮助。与此同时，也为实现子程序的共享提供了必要的资料。

【例 8.2】 编写一个子程序 EXCHASC，实现把寄存器 AL 中的一位十六进制数转换成其对应的 ASCII 码。

子程序如下：

```
; 子程序名:    EXCHASC
; 功　能:      把一位十六进制数转换为对应 ASCII 码
; 入口参数:    AL 的低 4 位=需转换的十六进制数码
; 出口参数:    AL=转换后得到的 ASCII 码
EXCHASC     PROC    NEAR
            AND     AL,0FH
            ADD     AL,30H
            CMP     AL,39H
            JBE     EXCHASC1
            ADD     AL, 7
EXCHASC1:   RET
EXCHASC     ENDP
```

说明：

在读了上述关于子程序 EXCHASC 的说明信息后，即使不熟悉或不了解这个子程序本身，也就能够方便地调用它了。

8.2　子程序的调用和返回

子程序调用指令和返回指令属于程序控制指令这一组。通常，调用指令用于由主程序转入子程序，返回指令用于由子程序返回主程序。由于程序代码可分为多个段，所以，像无条件转移指令一样，调用指令有段内调用和段间调用之分；与之相对应，返回指令也有段内返回和段间返回之分。把段内调用和段内返回称为近调用和近返回，把段间调用和段间返回称为远调用和远返回。在汇编语言中，子程序也有远近类型之分。

调用子程序有时也称为“转子”，即“转移到子程序入口处执行”。当要转去执行的子程序定义为段内调用时，为了转移到子程序入口处，只需修改 IP 寄存器的内容即可。当

要转去执行的子程序定义为段间调用时，意味着“调用”要转移到另一个段处执行，这时的子程序入口地址转移到新的段地址和偏移地址，此时不仅要修改寄存器 IP 的内容，还要修改寄存器 CS 的内容。

8.2.1 调用指令

调用指令 CALL 首先把子程序的返回地址（即 CALL 指令的下一条指令的地址）压入堆栈，以便执行完于程序后返回调用程序（主程序）继续往下执行。然后转移到子程序的入口地址去执行子程序。按照转移目标是否在同一段来分，调用指令分为段内调用和段间调用；按照获得转移目标地址的方式来分，调用指令分为直接调用和间接调用。下面介绍这四种调用指令，在汇编语言中，均用指令助记符 CALL 表示。

过程调用指令不影响标志。

1. 段内直接调用

段内直接调用是先修改堆栈指针，并将当前调用指令的下一条指令的地址（16 位）压入堆栈，再将当前的“下一条指令的地址”加上“子程序的入口地址”的 16 位的偏移量。其指令格式如下。

格式：

```
CALL  NEAR PTR  PROC_NAME
```

其中，

（1）PROC_NAME 为用 PROC/ENDP 定义的子程序名。

（2）当被调用的子程序具有 NEAR 属性并与调用指令 CALL 在同一个段内时，可用这种格式进行调用。当不引起歧义的情况下，NEAR PTR 可省略。

该指令进行的具体操作分解如下：

① SP←(SP)−2

② ((SP))←IP

③ lP←lP+disp

其中 disp 为地址差，它等于目标地址到 CALL 指令下一条指令开始地址的差，汇编程序在汇编时自动计算出 disp。在段内直接调用指令中，总用一个字表示 disp，所以转移范围可达−32768～+32767。

与无条件段内直接转移指令相似，段内直接调用指令的转移是相对转移，指令由操作码和地址差构成。段内直接调用指令只把返回地址的偏移部分压入堆栈保存，堆栈变化如图 8.1 所示。实际上，转移发生在同一段内，代码段寄存器 CS 的内容不发生变化，所以不必保存。然后把返回地址与子程序入口地址的差值（disp）加到指令指针 lP 上，使 IP 的内容为目标地址偏移，从而达到转移的目的。

【例 8.3】 设 SUBPROC 为一个子程序名，其属性为 NEAR 类型。写出其段内调用的相对寻址方式。

指令如下：

```
CALL  SUBPROC  或    CALL  SUBPROC  NEAR
```

2. 段内间接调用

段内间接调用也用于调用当前段内的子程序，它将当前的指令指针 IP 的内容压栈以

保存断点，然后把通用寄存器的内容或字存储单元中的内容送入 IP。其指令格式如下：

格式：

```
CALL    REG
或
CALL    WORD PTR    WADDR
```

其中，REG 为一通用寄存器，WADDR 为一变量或标号名。当 WADDR 无歧义地表示一个字存储单元时，WORD PTR 可以省略。当被调子程序具有 NEAR 属性并与调用指令 CALL 在同一个段内时，可以用这两种格式之一进行调用，也就是利用通用寄存器或字存储单元作为操作数进行段内间接调用。

该指令进行的具体操作分解如下：

① SP←(SP)−2

② ((SP))←IP

③ lP←(OPRD)

其中 OPRD 为通用寄存器 REG 或字存储单元 WADDR 中的内容。

该指令只是把返回地址的偏移部分压入堆栈保存，堆栈变化如图 8.1 所示。若是 16 位通用寄存器操作数，则把寄存器的内容送 IP；若是字存储器操作数，则把字存储单元之内容送 IP。

例如：

```
CALLAX
CALL    WORD PTR[AX]
CALL    VARW                    ; VARW 是字变量
```

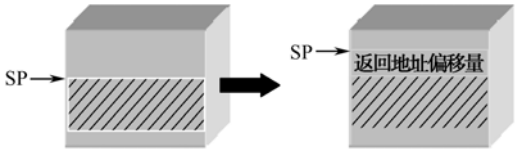


图 8.1　执行段内调用指令时的堆栈变化

3. 段间直接调用

段间直接调用指令用于调用其他代码段中的子程序。它首先将当前的代码段寄存器 CS 与指令指针 IP 的内容压栈以保存断点，然后把子程序所在的段基值送入 CS，偏移量送入 IP。其指令格式如下。

格式：

```
CALL  FAR  PTR  PROCNAME
```

其中，PROCNAME 为子程序名。要求被调子程序具有 FAR 属性，此调用指令为段间直接调用。当不引起歧义的情况下，FARPTR 可省略。

该指令进行的具体操作分解如下：

① SP←(SP)−2

② ((SP))←CS

③ SP←(SP)−2

④ ((SP))←IP

⑤ IP←过程入口地址的偏移

⑥ CS←过程入口地址的段值

段间直接调用指令与无条件段间直接转移指令相似，机器指令中含有转移目标地址。堆栈的变化如图 8.2 所示。

例如：

```
CALL    FAR  PTR  PROCSUB        ; PROCSUB 是远过程
CALL    PROCSUB                  ; PROCSUB 是远过程
```

汇编程序 MASM 能根据过程名所指定的被调用过程的类型决定采用段内直接调用指令还是段间直接调用指令。如果先调用后定义，那么为了调用远过程须在过程名前叫上类型说明符号“FAR PTR”。

4. 段间间接调用

段间间接调用也用于调用其他代码段中的子程序。它是将当前的代码段寄存器 CS 与指令指针 IP 的内容压栈以保存断点，然后把双字存储器操作数中的第一个字存储单元中的内容送入 IP，把第二个字存储单元中的内容送入 CS。其指令格式如下：

格式：

```
CALL    DWORD  PTR      DWADDR
```

其中，DWADDR 为一变量或标号名。当 DWADDR 无歧义地表示一个双字存储单元时，DWORD PTR 可以省略。当被调子程序具有 FAR 属性时，可以用这种格式进行调用，即利用双字存储单元作为操作数进行段间间接调用。

该指令进行的具体操作分解如下：

① SP←(SP)−2

② SP←(CS)

③ SP←(SP)−2

④ ((SP))←(IP)

⑤ IP←DWORD 的低字值　　; DWORD 为双字存储器操作数

⑥ CS←DWORD 的高字值

该指令把返回地址的段地址和偏移量分别压入堆栈保存（堆栈变化如图 8.2 所示），然后把双字存储器操作数的低字送 IP，把双字存储器操作数的高字送 CS，从而实现远转移。

例如：

```
CALL    DWORD  PTR [BX]
CALL    DVAR                     ; 设 DVAR 是双字变量
CALL    DWORD  PTR  TABLE[BX][SI]
```

对于间接调用，如果操作数是 16 位数，则汇编成段内间接调用，如果操作数是 32 位数，则汇编成段间间接调用。如果发生调用指令语句在先，有关变量定义伪指令语句在后的情况，则需要在调用语句中用 PTR 等操作符加以说明。

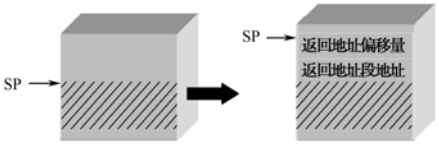


图 8.2　执行段内调用指令时的堆栈变化

8.2.2　返回指令

返回指令 RET 是子程序逻辑上的最后一条指令，也就是最后一条被执行的指令，它使子程序在完成任务后返回到调用它的 CALL 指令的后续指令处，即返回主程序处继续执行。根据对该子程序的调用是段内调用还是段间调用，其要实现的操作是不相同的。如果是段内调用，那么 RET 指令的功能只是把存放在堆栈里的返回地址送入 IP 寄存器中；如果是段间调用，那么 RET 指令的功能就是要把存放在堆栈里的返回地址和段地址，分别送 IP 寄存器和 CS 寄存器。返回指令不影响标志。

1. 段内返回

当返回指令所在的子程序具有 NEAR 属性时，此返回指令为段内返回。此返回指令是从堆栈中弹出一个字的内容，送入 IP。当 RET 带有操作数时，在完成上述的堆栈操作后，将此操作数加到 SP 上。其格式如下。

格式：

```
RET
    或
RET   N
```

其中，N 为操作数，为偶数。

该指令进行的具体操作分解如下：

① IP←(SP)

② SP←SP+2+(N)　　；如果带有操作数 N

如果不带有操作数 N，该指令只从堆栈弹出一个字，送到指令指针 IP。堆栈变化过程如图 8.3 所示。它与段内调用指令相对应。如果带有操作数 N，该指令先从堆栈弹出一个字作为返回地址，再额外根据 N 修改堆栈指针。所以该指令不仅能实现段内返回，而且能同时再修改堆栈指针。由于对堆栈的正常操作均是以字为单位，所以，N 的值一般应是偶数。

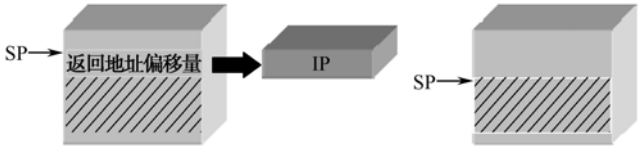


图 8.3　段内返回指令出栈示意图

2. 段间返回

当返回指令所在的子程序具有 FAR 属性时，此返回指令为段间返回。此返回指令是从堆栈中弹出两个字，将第一个字的内容送入 IP，将第二个字的内容送入 CS。同样，当 RET 带有操作数时，在完成上述的堆栈操作后，将此操作数加到 SP 上。

格式：

```
RET
    或
RET   N
```

其中，N 位操作数，为偶数。

该指令进行的具体操作分解如下：

① IP←(SP)

② SP←SP+2

③ CS←(SP)

④ SP←SP+2+(N)　　；如果带有操作数 N

该指令从堆栈弹出两个字，分别送到指令指针 IP 和代码段寄存器 CS。堆栈变化过程如图 8.4 所示。它与段间调用指令相对应。如果带有操作数 N，则再额外修改堆栈指针，即把 N 加到 SP。所以该指令不仅能实现段间返回，而且能同时再修改堆栈指针。

例如：

```
RET    4
```

设在执行上面的指令前 SP=C47EH，则在执行后 SP=C486H。

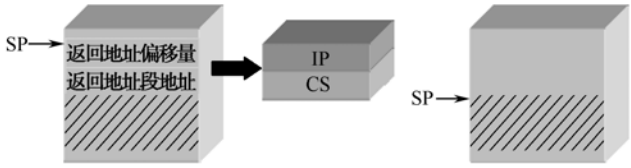


图 8.4　段间返回出栈示意图

注意：

尽管段内返回指令的助记符与段间返回指令的助记符是相同的，但它们的机器指令码是不同的。汇编程序 MASM 能根据 RET 指令所在过程的类型决定采用段内返回指令还是采用段间返回指令，只有当返回指令语句出现在远类型过程中时，MASM 才把它汇编成段间返回指令。

【例 8.4】　设一个主程序和一个子程序同在段内，和另一个子程序在不同的段中，写出此主程序分别调用这两个子程序的程序片段。

代码如下：

```
CODE1    SEGMENT
         …
START:
         …
         CALL     SUB1
         …
         CALL     FAR  PTR  SUB2
SUB1     PROC     NEAR
         …
         RET
SUB1     ENDP
CODE1    ENDS
CODE2    SEGMNET
         …
SUB2     PROC     FAR
```

```
        …
        RET
SUB2    ENDP
CODE2   ENDS
        END     START
```

说明：

若一个子程序既被段间调用又被段内调用，则其类型必须是 FAR。

8.2.3　现场保护和恢复

主程序和子程序执行过程中可能会共享一些相同的寄存器。如果不保护这些寄存器中的原有内容，子程序返回后，主程序将无法正确使用这些寄存器，这常常会导致主程序的错误。为此，在子程序调用过程中，要对有关寄存器或存储单元的内容进行保护与恢复。

寄存器的保护与恢复有两种方法。

（1）主程序中把需要保护的寄存器的内容压入堆栈和弹出堆栈。这种方法的优点是：在每次调用子程序时，只要把主程序所使用的寄存器压入堆栈。但缺点是：主程序每处调用都要保护与恢复，会增加程序的长度。

（2）在子程序一开始就把在子程序中要改变的寄存器内容压入堆栈，而在返回之前再恢复这些寄存器的内容。这种方法的优点是：在主程序中可方便地调用子程序，而无须考虑要把哪些寄存器压入堆栈，只需要在子程序中写一次压入和弹出堆栈指令组即可。这种方法是常用的方法。

若子程序 PROCSUB 中可能会改变 AX、BX、DX、DI、SI 这 5 个寄存器的内容，则在子程序的开始处应将这些寄存器的内容入栈保护，在子程序的返回指令前用出栈指令逐个恢复。实现方法如下：

```
PROCSUB   PROC
          PUSH    AX
          PUSH    BX
          PUSH    DX        ; 现场保护
          PUSH    DI
          PUSH    SI
          …                 ; 子程序工作部分
          POP     SI
          POP     DI
          POP     DX        ; 现场恢复
          POP     BX
          POP     AX
          RET               ; 返回断点处
PROCSUB   ENDP
```

若在主程序中保存寄存器，则一定在主程序中恢复；若在子程序中保存寄存器，则一定在子程序中恢复。这样可以增强主程序和子程序之间的相对独立性，减少相互依赖，使程序结构清楚，便于维护，减少错误。

注意：

（1）子程序中会改变标志寄存器中的标志，则可用 PUSHF 指令和 POPF 指令保护和恢复标志寄存器。

（2）在利用堆栈进行寄存器的保护和恢复时，要注意堆栈的“先进后出”特点。

（3）是否要保护入口参数寄存器，可以根据实际情况事先约定。一般出口寄存器不应保护

（4）在子程序中应尽量避免把普通存储单元作为临时变量使用，可以利用堆栈元素作为临时变量使用。

8.3 子程序与主程序间的参数传递

子程序一般都是完成某种特定功能的程序段。当一个主程序调用一个子程序时，通常都向子程序传递若干个数据让它来处理。当子程序处理完后，一般也向调用它的主程序传递处理结果，这种在主程序和子程序之间的信息传递为“参数传递”。

主程序向子程序传递的参数称为子程序的“入口参数”，子程序向调用它的主程序传递的参数称为子程序的“出口参数”。传递参数的多少反映程序模块间的耦合程度。根据实际情况，子程序可以只有入口参数或出口参数，也可以入口参数和出口参数都有，也可以入口参数和出口参数都没有。参数的具体内容可以是数据本身（传数值），也可以是数据的存储地址（传地址）。

参数传递的方法有多种：利用寄存器传递参数、利用堆栈传递参数、利用地址表传递参数和利用 CALL 后续区传递参数等。主程序与子程序间传递参数的方法是根据具体情况而事先约定好的，有时可能同时采用多种方法。

8.3.1 利用寄存器传递参数

利用寄存器传递参数就是把参数放在约定的寄存器中。这种方法的特点是：CPU 中的寄存器在任何程序中都是“可见”的，一个程序对某寄存器赋值后，在另一个程序中就能直接使用，所以，用寄存器传递参数最直接、简便，也是最常用的参数传递方法。但是，CPU 中寄存器的个数和容量都是非常有限，所以，该方法仅适用于传递较少的参数信息。

【例 8.5】 写一个判别字符是否为数字符的子程序。并利用该子程序把一个字符串中的所有数字符删除。

（1）分析问题

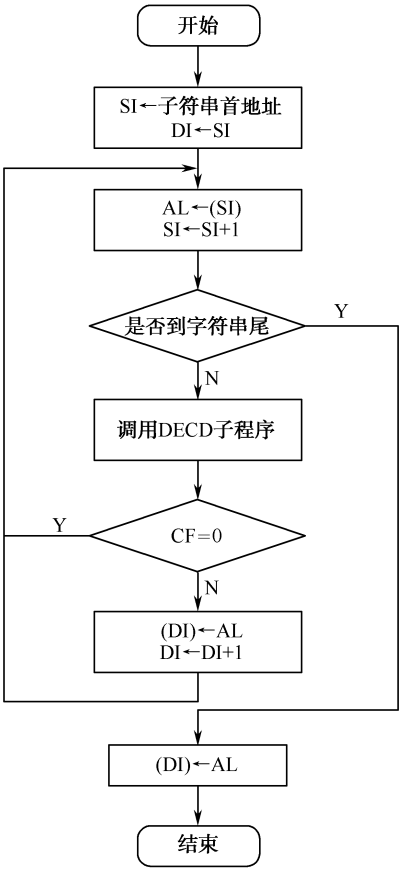
可以分别取字符串的的字符，并调用子程序来判断此字符是否为数字，如果为数字，则舍弃；如果不为数字，则保存。

（2）确定算法

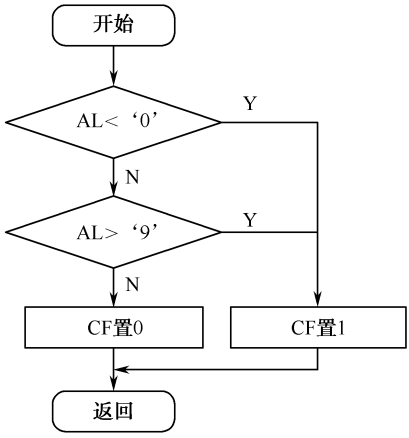
利用循环分别取出字符串的字符，调用子程序来判断此字符是否在“0”到“9”之间，如果在，则舍弃；如果不在，则保存此字符。

（3）流程图

流程图如图 8.5 所示。



（a）主程序



（b）DECD子程序

图 8.5　删除字符串中数字符的程序流程图

（4）确定汇编语言程序框架

该汇编语言程序的基本框架至少需要两个段：数据段和代码段。数据段中定义一个变量，即一个数组变量 STRING，为 DB 类型，使用寄存器 SI 定位，保存时使用 DI 定位。取出的每个字符存放在 AL 中。

（5）编写程序

程序如下：

```
DATA        SEGMENT
STRING      DB   'A8CA50-98sd02', 0              ; 假设的字符串
DATA        ENDS
CODE        SEGMENT
            ASSUME  CS: CODE, DS: DATA
; 子程序名:    DECD
; 功　能:      判别字符是否为数字符
; 入口参数:    AL=字符
; 出口参数:    CF 为 0，表示字符是数字符；否则字符是非数字符
DECD        PROC
            CMP     AL, '0'
```

```
            JB      DECD1
            CMP     AL, '9'
            JA      DECD1
            CLC                         ; 置 CF 为 0
            RET
DECD1:      STC                         ; 置 CF 为 1
            RET
DECD        ENDP
START:      MOV     AX, DATA
            MOV     DS, AX
            MOV     SI, OFFSET STRING   ; 置取指针
            MOV     DI, SI              ; 置存指针
NEXT:       MOV     AL, [SI]            ; 取一个字符
            INC     SI
            OR      AL, AL              ; 是否到字符串尾?
            JZ      OK                  ; 是，转
            CALL    DECD                ; 调用子程序
            JNC     NEXT                ; 是，不保存而处理下一个字符
            MOV     [DI], AL            ; 否，保存
            INC     DI
            JMP     NEXT                ; 处理下一个字符
OK:         MOV     [DI],AL
            MOV     AH,4CH
            INT     21H                 ; 返回 DOS 系统
CODE        ENDS
            END     START
```

说明：

上例中字体加黑的部分为子程序。入口参数使用 AL 传递，出口参数使用 CF 标志位传递。

8.3.2 利用堆栈传递参数

利用堆栈传递参数是指子程的入口参数和出口参数通过堆栈传递。如果使用堆栈传递入口参数，那么主程序在调用子程序之前，把需要传递的参数依次压入堆栈，子程序从堆栈中取入口参数。如果使用堆栈传递出口参数，那么子程序在返回前，把需要返回的参数存入堆栈，主程序在堆栈中取出口参数。

利用堆栈传递参数的优点是不占用寄存器，也无须使用额外的存储单元。但由于参数和子程序的返回地址混杂在一起，有时还要考虑保护寄存器，所以较为复杂。通常利用堆栈传递入口参数，而利用寄存器传递出口参数。这里要注意的是，从堆栈中读取数据与从堆栈中弹出数据是有区别的，从堆栈中读取数据并不改变堆栈的栈顶指针 SP，而从堆栈中弹出数据则需修改 SP。在使用堆栈传递参数时，返回指令 RET 应当使用带常数的返回指令，以便返回主程序后弹出提前压入堆栈的参数，恢复堆栈的原始状态。

【例 8.6】　改写例 8.5 程序，利用堆栈传递参数。

在主程序中，将待检测的字符压入堆栈，子程序从堆栈中读取此字符并进行判断，若为数字符，则将 CF 置 0；否则置 1。该汇编语言程序的基本框架至少需要 3 个段：堆栈段、数据段和代码段。堆栈段的存在是因为有参数传递、现场保护和恢复问题。程序的流程图如图 8.6 所示。

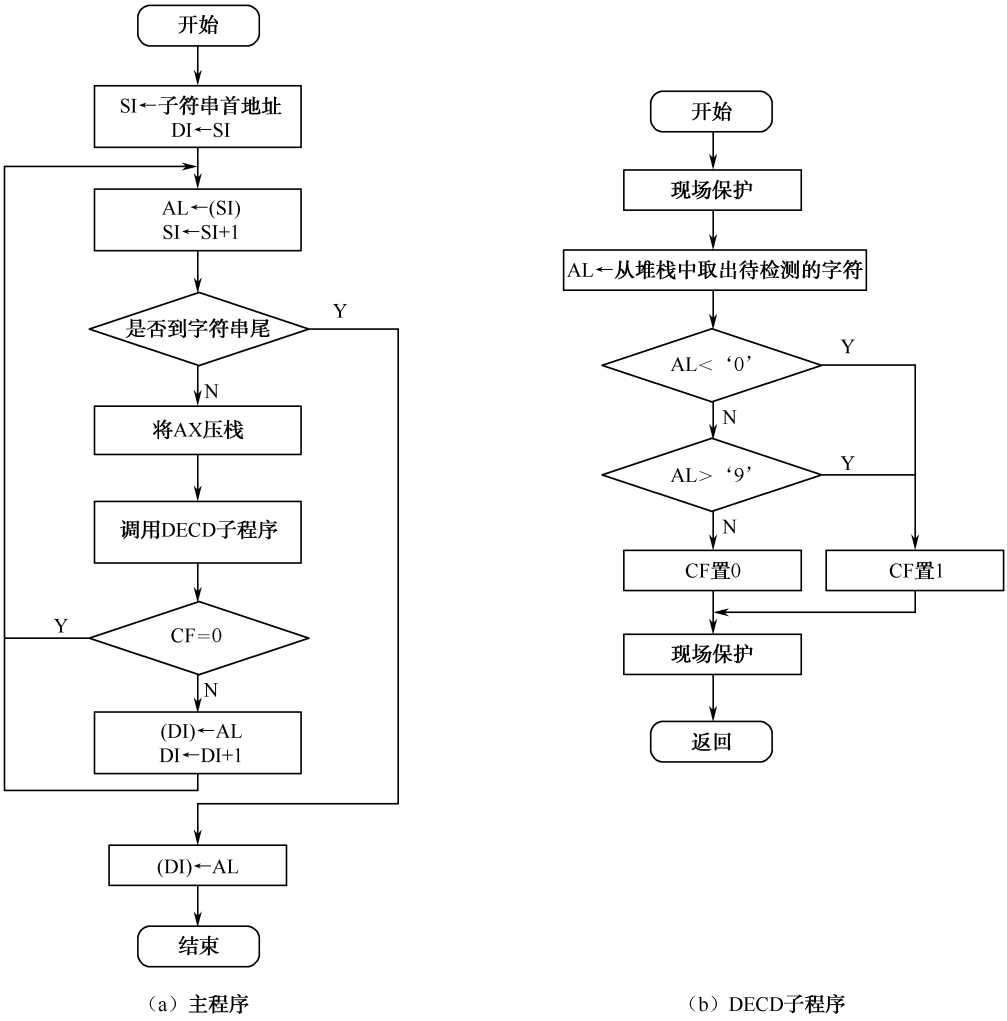


图 8.6　利用堆栈传递参数来实现删除字符串中数字符流程图

程序如下：

```
STACK       SEGMENT     PARA     STACK
            DW  20H     DUP(?)
STACK       ENDS
DATA        SEGMENT
STRING      DB   'A8CA50-98sd02', 0            ; 假设的字符串
DATA        ENDS
CODE        SEGMENT
```

```
                ASSUME  CS: CODE, DS: DATA, SS, STACK
; 子程序名:     DECD
; 功   能:      判别字符是否为数字符
; 入口参数:     堆栈字符
; 出口参数:     CF 为 0 表示字符是数字符，否则字符是非数字符
DECD            PROC
                PUSH    BP                  ; 将 BP 的原始内容压栈，保护现场
                MOV     BP, SP              ; 将 SP 的内容保存在 BP 中
                PUSH    SI                  ; SI 压入栈，SP 减 2
                PUSH    DI                  ; DI 压入栈，SP 减 2
                MOV     AX, [BP+4]          ; 从堆栈中取出待检测的字符
                CMP     AL, '0'
                JB      DECD1
                CMP     AL, '9'
                JA      DECD1
                CLC                         ; 置 CF 为 0
                JMP     OUT                 ; 跳转到 OUT
DECD1:          STC                         ; 置 CF 为 1
OUT:            POP     DI
                POP     SI
                POP     BP                  ; 恢复现场
                RET     2                   ; 返回主程序
DECD            ENDP
START:          MOV     AX, DATA
                MOV     DS, AX
                MOV     SI, OFFSET STRING   ; 置取指针
                MOV     DI, SI              ; 置存指针
NEXT:           MOV     AL, [SI]            ; 取一个字符
                INC     SI
                OR      AL, AL              ; 是否到字符串尾?
                JZ      OK                  ; 是，转
                PUSH    AX                  ; 将 AX 压栈
                CALL    DECD                ; 调用子程序
                JNC     NEXT                ; 是，不保存而处理下一个字符
                MOV     [DI], AL            ; 否，保存
                INC     DI
                JMP     NEXT                ; 处理下一个字符
OK:             MOV     [DI],AL
                MOV     AH,4CH
                INT     21H                 ; 返回 DOS 系统
CODE            ENDS
                END     START
```

说明：

例 8.6 中子程序被调用前后堆栈的变化情况如图 8.7 所示。子程序借助指针寄存器 BP，采用寄存器相对寻址方式取得入口参数，在子程序保存信息后，进行 BP←(SP)的操作。这时，在堆栈段中 BP、SP 有相同的偏移量，均指向堆栈顶部。入口参数与 BP 之间的相对距离是 4。所以使用下面的指令，就能分别从堆栈中取出入口参数：

```
MOV     AX, [BP+4]                    ; 将待检测的字符 0 送入 AX 中
```

另外要注意的是，当 BP 寄存器作为指针使用时，隐含使用的段寄存器是 SS，这正是堆栈段。

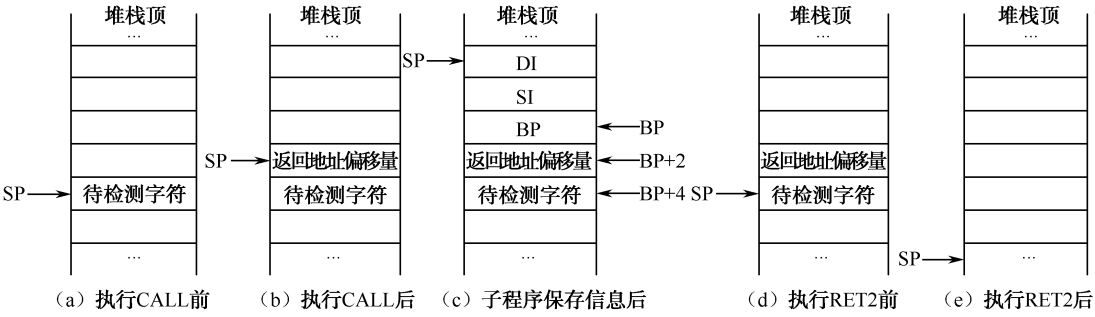


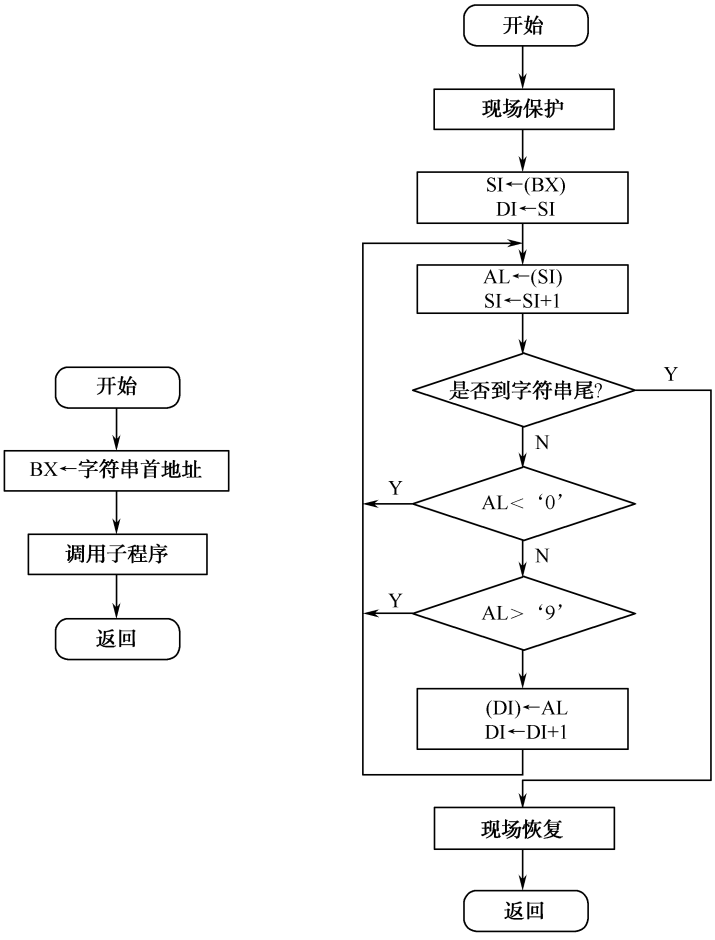
图 8.7　用堆栈传递参数时堆栈的变化

8.3.3　利用约定存储单元传递参数

在调用子程序时，有时需要向子程序传递大量数据。因受到寄存器容量的限制，这时就不能采用寄存器传递参数的方式，而要改用约定存储单元的传送方式。这种方式是主程序和子程序按照事先的约定，在指定的存储单元中进行数据交换，或者把参数存放在数据段的存储单元中，将参数区的首地址存放在某个寄存器中，然后用寄存器传递参数的办法，把这个地址传递给子程序。这种参数传递方法有点像情报人员之间联络的传递信息方式，一个向指定地点放情报，另一个从指定地点取情报。这种方法的优点是：在入口参数较多时很方便，而且不占用寄存器。当返回参数较多时，可用同样的方法传递参数，供主程序使用。另外子程序要处理的数据或进出的结果都有独立的存储单元，编写程序时不易出错。但这种方法要占用一定的存储单元，而且由于任何程序在任何时刻都可以修改这些数据，不利于模块化设计和信息隔离。

【例 8.7】　改写例 8.5 程序，利用约定存储单元传递参数。

可以将子程序设计成根据主程序传送过来的字符串首地址对字符串中各个字符循环做判断，当是数字符时舍弃，主程序将字符串首地址传送到 BX 寄存器中，程序流程图如图 8.8 所示。



（a）主程序　　（b）子程序

图 8.8　利用约定存储单元传递参数来实现删除字符串中数字符流程图

程序如下：

```
DATA        SEGMENT
STRING      DB   'A8CA50-98sd02', 0          ; 假设的字符串 STRING
DATA        ENDS
CODE        SEGMENT
            ASSUME  CS: CODE, DS: DATA
DECD        PROC
            PUSH    SI                      ; SI 压入栈，现场保护
            PUSH    DI                      ; DI 压入栈，现场保护
            PUSH    AX                      ; AX 压入栈，现场保护
            MOV     SI, [BX]                ; 将取字符指针偏移地址送入 SI 中
            MOV     DI, SI                  ; 将存字符指针偏移地址送入 DI 中
NEXT:       MOV     AL, [SI]                ; 取一个字符
            INC     SI
            OR      AL, AL                  ; 是否到字符串尾?
```

```
        JZ      OK                  ; 是，转
        CMP     AL, '0'
        JB      DECD1
        CMP     AL, '9'
        JA      DECD1
        JNC     NEXT                ; 是，不保存而处理下一个字符
DECD1:  MOV     [DI], AL            ; 否，保存
        INC     DI
        JMP     NEXT                ; 处理下一个字符
OK:     POP     AX                  ; 恢复现场
        POP     DI                  ; 恢复现场
        POP     SI                  ; 恢复现场
        RET                         ; 返回主程序
DECD    ENDP
START:  MOV     AX, DATA
        MOV     DS, AX
        MOV     BX, OFFSET  STRING  ; 将 STRING 首地址存到 BX 中
        CALL    DECD                ; 调用子程序
        MOV     AH,4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START
```

说明：

由例 8.7 可以看出，由于在子程序中没有直接访问变量名，而是通过地址访问变量的值，从而使子程序更通用。

8.4　子程序的嵌套与递归调用

主程序可以调用子程序，子程序也可以调用子程序，满足一定条件的子程序还可以实现递归调用。

8.4.1　子程序的嵌套调用

子程序也可以作为主程序去调用另一个子程序，这种情况就称为子程序的嵌套。嵌套的层数不限，其层数称为嵌套深度，图 8.9 表示的是嵌套深度为 3 时的子程序嵌套情况。但由于子程序的调用需要在堆栈中保存返回地址及寄存器等数据，因此实际上受限于开设的堆栈空间。

嵌套子程序的设计并没有什么特殊要求，除子程序的调用和返回应正确使用指令外，要注意寄存器的保存和恢复，以避免各层次子程序之间因寄存器冲突而出错的情况发生。如果程序中使用了堆栈，如使用堆栈来传送参数等，则对堆栈的操作要格外小心，避免发生因堆栈使用中的问题而造成子程序不能正确返回的错误。

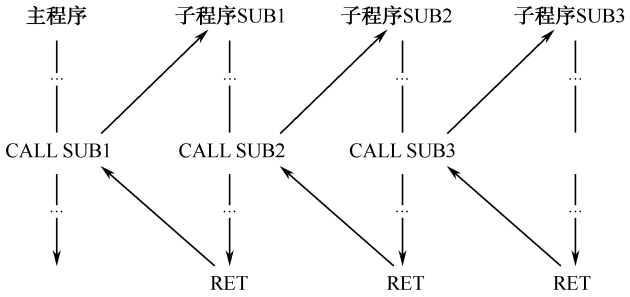


图 8.9　子程序的嵌套示意图

8.4.2　子程序的递归调用

当子程序嵌套地调用自身时，称为递归调用，含有递归调用的子程序称为递归子程序，或称递归过程。不是所有子程序都可以被递归调用，递归子程序的设计必须保证每次调用都不破坏以前调用时所用的参数与中间结果，递归子程序又必须有递归结束的条件，以防止递归调用的无限嵌套。为此，每次调用时将要用到的参数及中间结果都不能存放在相同的存储区中，所以一般将输入参数、寄存器内容及中间结果都存放在堆栈中。这里要注意的是，递归深度受堆栈空间的限制。

若将一次调用时所要保存的信息称为一帧信息，那么一帧信息可以包括调用时所传递的入口参数、出口参数、返回地址，以及子程序中要用到的有关寄存器的内容和局部变量。每次调用时，将一帧信息压入堆栈，每次返回时，则从堆栈中弹出一帧信息。

【例 8.8】　编写一个递归子程序，实现 N!的运算，其中 0≤N≤6。

（1）分析问题

阶乘运算规则如下：

N!=N(N−l)!=N(N−l)(N−2)!= …

0!=l

如果要求 N 的阶乘，就是反复求 N(N−l)!=N(N−1)(N−2)!，可以用一个递归子程序来实现。每次递归调用时，应将调用参数减 1，当调用参数为 0 时，应停止递归调用，且有 0!=l 的中间结果。假设求 N！使用 F(N)来表示，则：

F(N)=N· F(N−l)

=N(N−l)· F (N−2)

=N(N−l)(N−2)· F (N−3)

=N(N−l)(N−2)· F (N−3)…F(0)

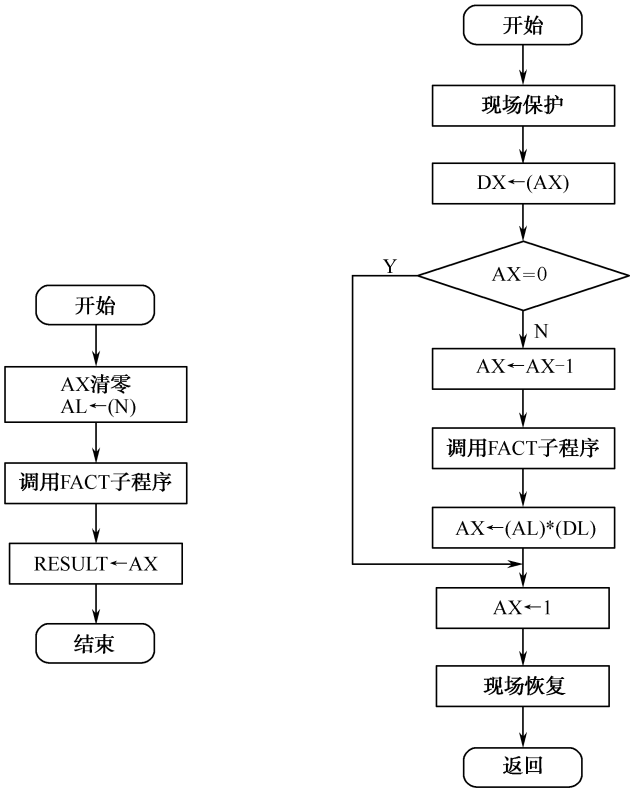
=N(N−l)(N−2)· F(N−3)…1

（2）确定算法

可以采用递归子程序来实现阶乘的运算。因为每次递归调用时参数都压入堆栈中，当 N 减为 0 而程序开始返回时，应按嵌套的方式逐层返回，并逐层取出相应的调用参数。子程序的入口参数存放在 AX 中，出口参数也存放在 AX 中。

（3）流程图

流程图如图 8.10 所示。



（a）主程序　　（b）FACT子程序

图 8.10　删除字符串中数字符的程序流程图

（4）确定汇编语言程序框架

该汇编语言程序应该包含有 3 个段：堆栈段、数据段和代码段。堆栈段中至少需定义两个变量：一个数组缓冲区，占 20H 个字。数据段中至少需定义两个变量：一个是数 N，选用 DB 类型；另一个是函数结果变量 RESULT，选用 DW 类型。数 N 也为循环次数，计数器用 AX。

（5）编写程序

程序如下：

```
DATA     SEGMENT
N        DB   6
RESULT   DW   ?
DATA     ENDS
STACK    SEGMENT      PARA      STACK
         DW        20H  DUP(0)
STACK    ENDS
CODE     SEGMENT
         ASSUME CS:CODE, DS:DATA, SS:STACK
```

```
START:  MOV     AX, DATA
        MOV     DS, AX
        XOR     AX, AX                  ; AX 清零
        MOV     AL, N                   ; 将要计算的阶乘数送到 AL 中
        CALL    FACT                    ; 调用子程序
        MOV     RESULT, AX
        MOV     AH, 4CH
        INT     21H
; 子程序名:FACT
; 功能：计算 N!
; 入口参数：(AX)=N
; 出口参数：(AX)=N!
FACT    PROC
        PUSH    DX                      ; 现场保护
        MOV     DX, AX
        CMP     AX, 0                   ;  N 为 0?
        JZ      DONE                    ; 为 0 则转到 DONE
        DEC     AX                      ; 否，则 N−l
        CALL    FACT                    ; 求(N−l)!
        MUL     DL                      ;  N(N−l)!
        POP     DX
        RET
DONE:   MOV     AX,1                    ;  0! =1
        POP     DX                      ; 现场恢复
        RET
FACT    ENDP
CODE    ENDS
        END     START
```

说明：

在编写嵌套或递归子程序时，堆栈的使用是十分频繁的。由于堆栈区域是在堆栈定义时就确定的，因而堆栈工作过程中有可能产生溢出。堆栈溢出有两种可能发生的情况：如堆栈已满，但还想再存入信息，这种情况称为堆栈上溢；另一种情况是堆栈已空，但还想再取出信息，这种情况称为堆栈下溢。上溢或下溢都是不允许的。因此在编写程序时，如果估计可能发生堆栈溢出，则应在程序中采取保护措施。这时，可先给 SP 规定上、下限，在进栈或出栈操作前先将 SP 和边界值进行比较。如果溢出，则进行溢出处理，以避免发生破坏其他存储区或程序出错。

习题

1．汇编语言的子程序是如何定义的。所使用的关键字有哪些?

2．为了编写具有良好风格的子程序，一般需要书写哪些重要的说明性信息?

3．子程序定义的方法及注意事项。

4．子程序段内调用和段间调用在程序设计时有何不同?

5．调用指令 CALL 和转移指令 JMP 之间有什么区别？

6．段内和段间子程序调用指令 CALL 有什么区别？

7．子程序返回指令 RET 的功能能否用 JMP 指令来模拟？若可以，请用段内子程序的返回加以说明；否则，说明理由。

8．子程序返回指令“RET 4”的功能是将返回数值“4”给调用程序吗？若不是，哪其作用是什么？

9．在子程序中要使其所用寄存器对调用者都是透明的，请举例说明达到其目的的方法。

10．编写一个子程序，根据参数实现大写字母转换成小写、小写字母转换成大写的大小写互换功能。

11．设某数组中含有 8 个有符号字节数，求其中最大值和最小值的差，并存放在变量 RESULT 中，其中查找最大值和最小值的程序分别使用子程序来实现，并且分别使用寄存器传递参数、堆栈传递参数来实现。

12．设有 12 个学生的成绩，分别为 89，64，65，73，89，77，100，67，90，83，72 和 67。要求编写一个子程序，分别统计 60～69，70～79，80～89，90～99 和 100 分的人数，并且存放到对应的 P6、P7、P8、P9 和 P10 的单元中。

13．编写一个子程序实现：S=1！+2！+3！+…+8!

14．编写程序，实现斐波那契兔子问题。假定小兔子一个月后可以长成大兔子，2 个月及之后的大兔子每个月都会生出一对小兔子。如果假定年初养了一个小兔子，问到年底将有多少对兔子。提示，程序的具体实现采用递归子程序的方法。

15．请按如下程序说明设计一个子程序。

（1）功能：计算指数函数 X^n 的值。

（2）入口参数：AL=X 的值，AH=n 的值。

（3）出口参数：BX。

第 9 章

输入/输出与中断程序设计

输入/输出功能是计算机的重要组成部分，是人机交互功能的主要承担者。输入/输出设备（I/O 设备）是指在计算机系统中除 CPU、内存储器等之外的设备。这些设备都有独立的芯片，有些芯片也是微处理器。计算机主体部分与外部设备之间有大量的信息需要传送，如数据、程序、现场采集的信号数据等，都需要通过输入设备输入至计算机由 CPU 进行处理，同时 CPU 处理的结果和各种控制信号也需要通过输出设备输出，以实现显示或各种控制操作。

输入/输出设备与主机之间的数据交换方式概况起来有程序、中断、DMA、通道和处理机几种方式，无论哪种方式都要用到输入/输出指令编写程序。

本章主要介绍输入/输出与中断的基本概念，以及输入/输出程序和中断处理程序设计。

9.1 输入和输出的基本概念

在计算机系统中，外部设备并不像内存储器那样可以直接与系统总线相连，它必须通过输入/输出（I/O）接口才能与系统总线相连，如打印机通过打印接口与系统总线相连，显示器通过显示控制器和系统总线相连，其主要原因是内存储器速度快且种类较少，而外部设备种类繁多且工作速度相差很大。

从程序设计角度来看，接口由一组寄存器组成，是完成输入/输出的桥梁。程序利用 I/O 指令，存取接口上的寄存器，从而获得外部设备的状态信息，控制外部设备的动作，从而实现输入/输出。

9.1.1 I/O 端口地址

计算机系统中 I/O 设备通过 I/O 接口与系统总线相连，如图 9.1 所示，而每个接口包含一组寄存器，分别是数据寄存器、状态寄存器、控制寄存器等，如图 9.2 所示。其中数据寄存器、状态寄存器、控制寄存器的功能如下。

（1）数据寄存器对来自或者送往 CPU 和内存的数据起缓冲作用。数据输出寄存器锁存 CPU 送出的数据信息，以提供给外设；数据输入寄存器暂存由外设传递给主机的数

据信息。根据不同的需要，在接口电路中还可以设置不同的数据寄存器，从一个到几十个不等。

（2）状态寄存器存放外围设备或者接口部件本身状态。外设通过状态寄存器存放向 CPU 提供的可查询的外设状态信息，CPU 可通过数据线读回，并根据外设的状态信息采取相应措施。

（3）命令寄存器存放 CPU 发出的命令，以便控制接口和设备动作。命令寄存器接受来自 CPU 的控制命令字，并将它们转换为外设可识别的操作命令。

CPU 通过访问这些寄存器和外部设备进行信息交换。为了能访问到这些寄存器，一般就像给内存储单元编址一样给这些寄存器编址。根据具体情况可以一个寄存器编一个地址，也可以是多个寄存器编一个地址。相应的寄存器或寄存器组被称为 I/O 端口，相应的地址被称为 I/O 端口地址或 I/O 端口号。每个 I/O 端口对应一个唯一的 I/O 端口地址。

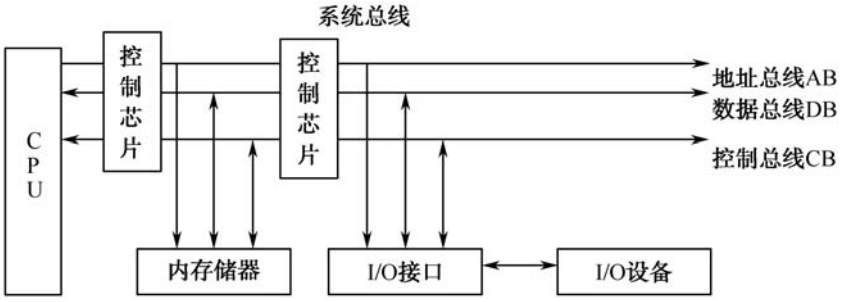


图 9.1　计算机系统基本结构

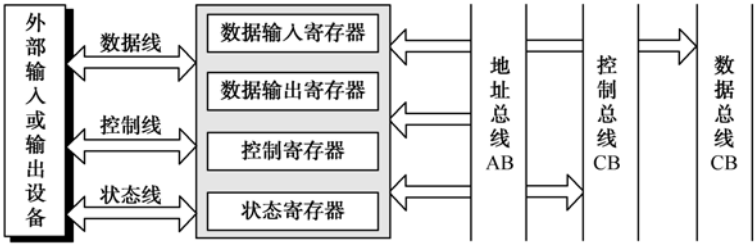


图 9.2　外设通过 I/O 接口和系统总线相连

9.1.2　I/O 端口编址

有的计算机内存储器和 I/O 端口是统一编址的，有的计算机内存储器和 I/O 端口则是分开编址的。将 I/O 端口地址和存储单元地址统一编址称为存储器映像编址方式。将内存储器和 I/O 端口是分开编址称为 I/O 端口独立编址。

1. 存储器映像编址方式

存储器映像编址方式将 I/O 端口和存储器单元同等对待，统一编址。相当于给每个 I/O 端口分配一个存储器地址，即 I/O 端口地址空间是存储器地址空间的一部分，如图 9.3 所示。对端口的访问，就像访问一个存储单元一样。所以对存储器的各种寻址方式，也同样

适用于对 I/O 端口的访问。

采用存储器映像编址的优点在于简化指令系统，无须专门的 I/O 指令。对 CPU 来说，对存储器的访问和对端口的访问没有任何区别。因此，程序设计灵活方便。缺点是/O 端口地址占用了一部分存储器地址空间，而且 I/O 指令码长，执行速度慢。Motorola 公司生产的各档微处理器就是采用这种 I/O 端口编址方式，如 MC6800/68000、6502 系列等。

2. 端口独立编址方式

端口独立编址又称为端口单独编址，I/O 设备的地址空间和存储器地址空间是两个独立的空间，如图 9.4 所示。CPU 使用专门的 I/O 指令来实现数据传送。在 Intel 的 80x86 系列处理器 CPU 系统中，I/O 端口地址和内存储单元的地址是各自独立的，分别占两个不同的地址空间。8086/8088 提供的 I/O 端口地址空间达 64KB，因而可接 64KB 个 8 位端口，或 32K 个 16 位端口，也就是有 65536 个字节端口，或 32768 个字端口。但实际上，PC 及其兼容机一般只使用了 0～3FFH 之间的 1/O 端口地址，只占整个 I/O 端口地址空间的很小一部分。在此种编址模式下，为了访问 I/O 端口，就必须使用专门的输入指令和输出指令。I/O 端口不分段，直接把输入/输出指令中给出的端口地址传送给地址总线。这样，在地址总线上就可能出现相同的端口地址和存储单元地址。为了区别地址总线上的地址是访问存储单元还是访问 I/O 端口，8086/8088 CPU 芯片有一个控制线 M/$\overline{IO}$（8086 芯片）或 $\overline{M}$ /IO（8088 芯片）。当 M/$\overline{IO}$=1 时，访问存储器。当 M/$\overline{IO}$=0 时，访问 I/O 端口。

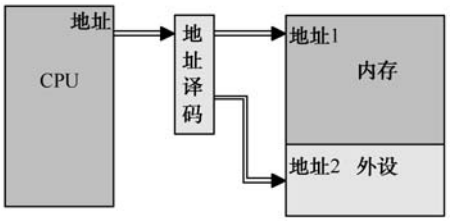


图 9.3 存储器映像编制

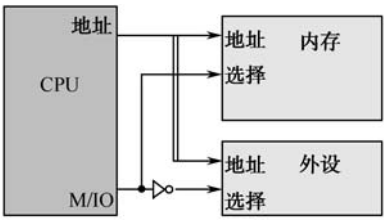


图 9.4 端口独立编址

在 8086 微机中，I/O 端口编址在一个独立的地址空间，具体 I/O 端口地址分配如表 9.1 所示。

表 9.1 部分端口地址分配

端口地址	外设、接口或控制器	端口地址	外设、接口或控制器
00～0F	DMA 控制器 8237A	2F8～2FE	2 号串行口(COM2)
20～3F	可编程中断控制器 8259A	320～324	硬盘控制器
40～SF	可编程中断定时/计数器	366～36F	PC 网络
60～63	8255A PPI 可编程并行接口	372～377	软盘适配器
70～71	CMOS RAM	378～37A	2 号并行口(LPT1 打印机)
81～8F	DMA 页表地址寄存器	380～38F	SDLC 及 BSC 通信
93～9F	DMA 控制器	390～393	Cluster 适配器
A0～A1	可编程中断控制器 2	3A0～3AF	BSC 通信
C0～CE	DMA 通道，内存/传输地址寄存器	3B0～3BF	MDA 视频寄存器

续表

端口地址	外设、接口或控制器	端口地址	外设、接口或控制器
F0～FF	协处理器	3BC～3BE	1 号并行口
170～1F7	硬盘控制器	3C0～3CF	EGA/VGA 视频寄存器
200～20F	游戏控制端口	3D0～3D7	CGA 视频寄存器
278～27A	3 号并行口（LPT2 打印机）	3F0～3F7	软盘控制寄存器
2EO～2E3	EGA/VGA 使用	3F8～3FE	1 号串行口(COMl)

端口独立编址的优点是不占用存储器地址空间，但它的缺点是需专门的 I/O 指令。

9.1.3　I/O 指令

I/O 指令是 CPU 与外部设备进行通信的最基本途径，由于 8086/8088 的 I/O 端口地址和内存储单元地址是独立的，所以要用专门的 I/O 指令来存取端口上的寄存器，也就是说要用专门的 I/O 指令进行输入/输出。

1. 输入指令

输入指令是将指定端口的内容（字或字节）输入（读）到寄存器 AL 或 AX 中。其格式如下。

格式：

```
IN    AL,PORT          ; 字节
IN    AX,PORT          ; 字
      或
IN    AL,DX            ; 字节
IN    AX,DX            ; 字
```

其中，

（1）PORT 表示的是端口地址。前两种是直接端口寻址，PORT 是一个 8 位立即数，大小不得超过 255。

（2）后两种是间接端口寻址，端口地址在寄存器 DX 中。当端口地址超过 255 时，只能采用 DX 间接端口寻址。

例如：

```
IN      AL,12H              ; 直接端口寻址
MOV     DX,1DCH
IN      AX,DX               ; 间接端口寻址
```

上述指令中，第一条指令是直接寻址，从 12H 输入一个字节送入 AL。执行第二条和第三条指令时，相当于从 1DCH 输入一个字节送往 AL，从 1DDH 输入一个字节送往 AH。

2. 输出指令

输出指令将寄存器 AL 或 AX 的内容（字或字节）输出（写）到指定端口中。其指令格式如下。

格式：

```
OUT     PORT,AL             ; 字节
OUT     PORT,AX             ; 字
```

```
        或
OUT     DX,AL                    ; 字节
OUT     DX,AX                    ; 字
```

其中，

（1）PORT 表示的是端口地址。前两种是直接端口寻址，PORT 是一个 8 位立即数，大小不得超过 255。

（2）后两种是间接端口寻址，端口地址在寄存器 DX 中。当端口地址超过 255 时，只能采用 DX 间接端口寻址。

例如

```
MOV     AL,23H
OUT     40H,AL                   ; 直接端口寻址
MOV     AX,0D234H
MOV     DX,1EEH
OUT     DX,AX                    ; 间接端口寻址
```

上述程序的第一条和第二条指令是向端口 40H 输出 23H; 后三条指令是间接端口寻址，向端口 1EEH 输出 34H，向端口 1EFH 输出 0D2H。也就是说，当向端口 N 输出一个字时，相当于向端口 N 输出 AL 中的内容和向端口 N+l 输出 AH 中的内容。

I/O 指令都属于数据传送指令，它们指定使用寄存器 AL 或 AX。在程序设计时是用 IN 还是 OUT 指令，要根据设备及端口类型来决定，若为输入设备的数据端口，则使用 IN 指令；若为输出设备的数据端口，则使用 OUT 指令。

注意：

（1）IN 指令的目的操作数和 OUT 指令的源操作数只能使用寄存器 AL 或 AX，若要将 I/O 端口中的内容输入到其他寄存器或者内存单元中，只能先输入到这两个寄存器中，然后再传送到其他寄存器或者内存单元中。

（2）I/O 指令中的端口号或者寄存器 DX 的内容均为端口的地址，而传送的数据则是端口中的内容。

（3）I/O 指令有两种寻址方式，一种为直接端口寻址，另一种为间接端口寻址，间接端口寻址方式比较灵活。因为在程序中只要修改 DX 的内容，就可以用同一条输入/输出指令去访问不同的端口，实现可变端口的输入/输出。

(4) 端口 0~255 既可以用直接端口寻址，也可以用间接端口寻址。

9.2 数据传送方式

CPU 与外设之间交换的信息包括数据信息、控制信息和状态信息。尽管这三种信息具有不同的性质，但它们都通过 IN 和 OUT 指令在数据总线上进行传送，所以通常采用分配不同端口的方法将它们加以区别。

数据信息是 CPU 和外设真正要交换的信息。数据信息通常为 8 位或 16 位，可分为各种不同类型。不同的外设要传送的数据类型也是不同的。

控制信息输出到 I/O 接口，告诉接口和设备要做什么工作。

状态信息是从接口输入的，表示 I/O 设备当前的状态。在输入数据前，通常要先取得

表示设备是否已准备好的状态信息，在输出数据前，往往要先取得表示设备是否忙的状态信息。

系统中各种数据的传递方式主要划分为程序控制方式、中断控制方式、DMA 方式、通道和 I/O 处理机方式四种。

9.2.1　程序控制方式

程序控制方式是 I/O 操作的一种最简单的控制方式，它要求 I/O 操作在 CPU 直接控制下执行，即与 I/O 设备有关的数据传输操作都要求 CPU 执行程序来实现。这种方式的主要特点是与外设交换信息完全在用户程序的控制下进行，在哪个端口何时进行输入/输出操作均在程序中反映出来。

程序控制方式可分为立即传送和程序查询两种方式。

1. 无条件传送方式

无条件传送方式也称为立即传送方式，它是最简单的一种输入/输出传送方式。它不需要查询外设的状态，即认为外设总是处于准备好或空闲状态，可以直接使用 IN 或 OUT 指令实现数据传送。这种方式只要在指令中指明端口地址，就可选择指定外设进行输入/输出。微机系统中的一些简单的外设，如开关、继电器、数码管、发光二极管等，在它们工作时，可以认为输入设备已随时准备好向 CPU 提供数据，而输出设备也随时准备好接收 CPU 送来的数据，这样，在 CPU 需要同外设交换信息时，就能够用 IN 或 OUT 指令直接对这些外设进行输入/输出操作。

无条件传送方式使用方便，但要求外设工作速度能与 CPU 同步，否则就可能出错。例如，在外设还没有准备好的情况下，用 IN 指令得到的数据就可能是不正确的数据。

2. 程序查询方法

程序查询方式是在要作输入/输出操作时，首先查询外设的状态，确认可进行输入/输出操作后，再用输入/输出指令完成本次的输入/输出操作。程序查询方式用于 CPU 与外设不同步的情况。在输入之前，要查询外设的数据是否已准备好，直到外设把数据准备好后才开始输入；输出之前，查询外设是否“忙”，若“不忙”，则可以输出，否则继续查询，直到不“忙”。也就是说，要等待到外设准备好时才能输入或输出数据，而通常外设速度远远慢于 CPU 速度，于是查询过程将花费大量的时间。

为了采用查询方式输入或输出，相应的外设接口不仅要有数据寄存器，而且还要有状态寄存器，有些外设还要有控制寄存器。其中数据寄存器用来存放要传送的数据，状态寄存器用来存放表示设备所处的状态的信息。通常状态寄存器中有一个“就绪（Ready）”位和一个“忙（Busy）”位来反映外设是否准备好。

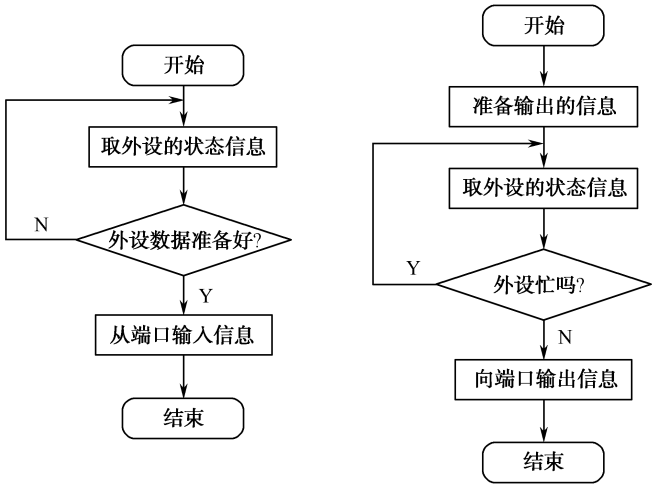
使用程序查询方法需要以下几个步骤才能实现输入/输出操作（流程图如图 9.5 所示）。

（1）获得状态信息。

（2）判断是否可以进行新的输入/输出操作。对输入操作，查询是否有新的信息送入端口（即准备好否？）。对输出操作，查询是否已完成上次的输出任务（即设备是否在忙？）。如果已准备好或设备空闲，方可进行下一步；否则返回第一步。

（3）执行一条输入/输出指令，从连接输入设备的端口取回一个字或字节的信息，或向

连接输出设备的端口送去一个字或字节的信息。



（a）查询方式输入操作流程图　　（b）查询方式输出操作流程图

图 9.5　查询方式输入/输出

在实际应用中，为防止设备因某种原因发生故障而无法就绪或空闲，从而导致 CPU 在无限循环之中，通常都设计一个等待超时值，其值随设备而定。一旦设备在规定时间内还无法就绪或空闲，也中止循环查询过程。如此，其流程图如图 9.6 所示。大多数情况下，等待超时值用查询次数表示，每查询一次，查询次数减 1，如果查询次数减到 0，那么查询等待也就结束。

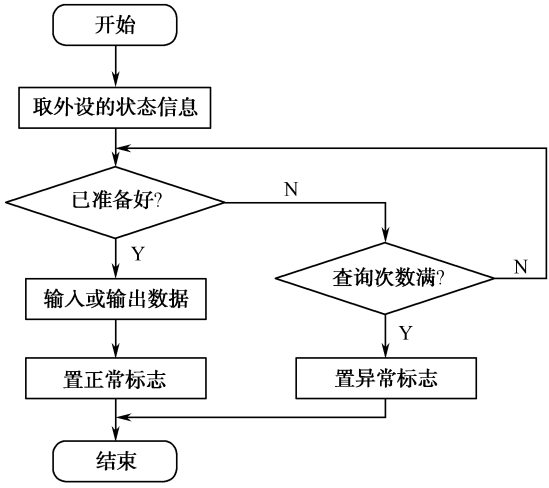


图 9.6　查询方式的一般流程图

有时系统中同时有几个设备要求输入/输出数据，那么对每个设备都可编写一段执行输入/输出数据的程序。然后轮流查询这些设备的状态寄存器中的就绪位，当某一设备准备好允许输入或输出数据时，就调用这个设备的 I/O 程序完成数据传送，否则依次查询下一设备是否准备好。

查询方式的优点是：软硬件实现比较简单；当同时查询多个外设时，可以由程序安排查询的先后次序。缺点是浪费了 CPU 原本可执行大量指令的时间。

【例 9.1】　编写一个子程序，使用查询方式打印一个字符。

（1）分析问题

打印机是通过打印接口连入系统，打印接口具有传递打印命令、打印数据和返回打印机状态三个功能，相应地应该包含控制寄存器、数据寄存器和状态寄存器。各接口定义如图 9.7 所示。

上述三个寄存器有各自的端口地址，并且三个端口地址是连续的。设数据寄存器端口地址是 378H，状态寄存器端口地址是 379H，控制寄存器端口地址是 37AH。在系统加电初始化期间，数据寄存器端口地址被保存到 BIOS 数据区。

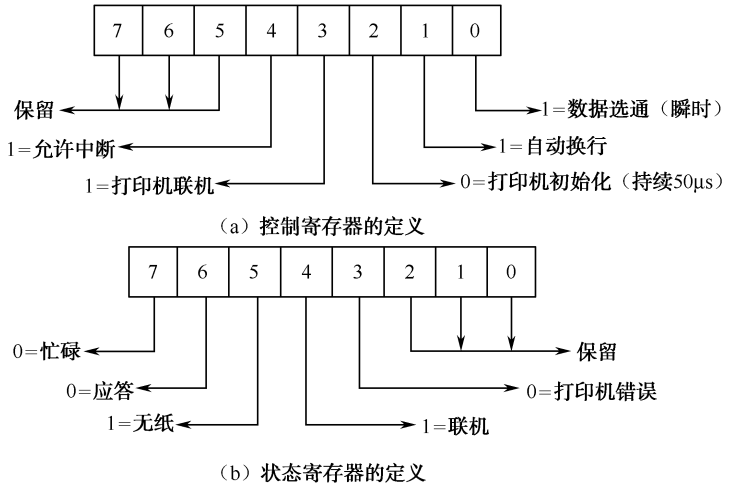


图 9.7　打印机的状态寄存器和控制寄存器各位的定义

若知道三个寄存器的端口地址，根据端口定义，就可以采用根据查询方式来进行打印流程作业。

（2）确定算法

首先准备好打印数据（存放到打印机数据寄存器中），但此时打印机并未接受，随后读取状态信息，将 DX 指向状态寄存器端口，读取状态信息，判打印机是否忙碌，一直等到打印机不忙碌时才将 DX 指向控制寄存器端口，向打印机发出选通命令打印相应的字符。

（3）画流程图

流程图如图 9.8 所示。

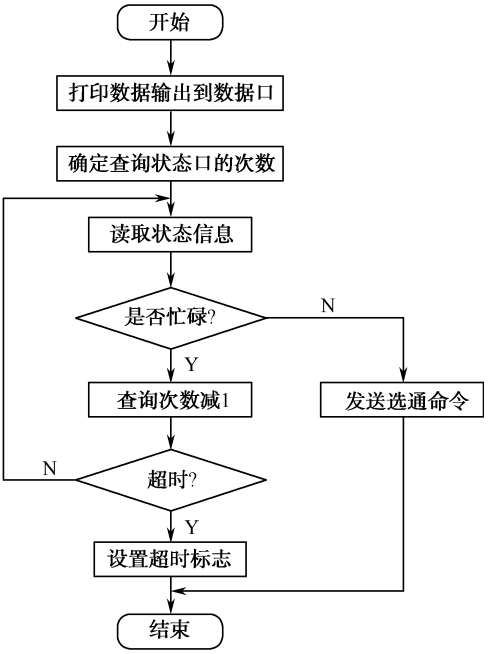


图 9.8　查询方式打印的程序流程图

（4）编写程序

程序如下：

```
; 子程序名：PROC
; 打印一个字符
; 入口参数：    DX=数据寄存器端口地址
;               BL=超时参数
;               AL=打印字符的 ASCII 代码
; 出口参数：    AH=打印机状态，各位定义如下：
;               位 0：表示超时，即超过规定的查询次数
;               位 1：保留未用
;               位 2：保留未用
;               位 3：1 表示出错
;               位 4：1 表示联机
;               位 5：1 表示无纸
;               位 6：1 表示应答
;               位 7：0 表示忙碌
PRINT   PROC
        PUSH    DX              ; 现场保护
        PUSH    AX              ; 现场保护
        OUT     DX,AL           ; 输出打印数据
        INC     DX              ; 使 DX 含状态寄存器端口地址
WAIT:   XOR     CX,CX           ; 设置一个超时参数单位，表示查询 65536 次
WAIT1:  IN      AL,DX           ; 读取状态信息
        MOV     AH,AL           ; 状态信息保存到 AH
        TEST    AL,80H          ; 检测是否忙碌
```

```
        JNZ     NEXT            ; 若不忙碌，则转到 NEXT
        LOOP    WAIT1           ; 继续查询
        DEC     BL              ; 超时参数减 1
        JNZ     WAIT            ; 未超时，继续查询
        AND     AH,0F8H         ; 已超时，去掉状态信息中的无用位
        OR      AH,1            ; 置超时标志
        JMP     EXIT            ; 转结束
NEXT:   INC     DX              ; 不忙，使 DX 含控制器端口地址
        MOV     AL,0DH          ; 准备选通命令
        OUT     DX,AL           ; 选通
        MOV     AL,0CH          ; 准备复位选通命令
        JMP     $+2             ; 延时
        OUT     DX,AL           ; 复位选通位
        AND     AH,0F8H         ; 去掉状态信息中的不用位
EXIT:   XOR     AH,48H          ; 使复位后状态信息中有关位符合要求
        POP     DX
        MOV     AL,DL           ; 恢复 AL 寄存器值
        POP     DX
        RET
PRINT   ENDP
```

说明：

（1）例 9.1 中使用了一个超时参数，一个超时参数表示查询 65536 次，应用中的超时参数应根据系统的具体情况而定。

（2）在发出选通命令和复位选通之间使用了一条转移指令“JMP $+2”，该指令起到延时作用，以保证有足够的选通时间。

（3）为了使（AH）寄存器中含有规定的出口参数，要将应答位和错误应答位的值取反，所以在子程序结束之前，用异或指令将这两位取反。

9.2.2　中断控制方式

程序控制方式在程序设计时比较简单，但是实际应用受限。例如，立即传送方式，完全不能反映设备实时状况；而程序查询方式，CPU 与外设串行工作，浪费 CPU 时间。为了提高 CPU 的效率，增加实时处理能力，可采用中断控制方式。当外设准备好时，外设向 CPU 发出中断请求，CPU 转入中断处理程序，完成输入/输出工作。

中断是一种使 CPU 挂起正在执行的程序而转去处理特殊事件的操作。这些引起中断的事件称为中断源。它们可能是来自外设的输入/输出请求，例如，由按键引起的键盘中断，由串行口接收到信息引起的串行口中断等，也可能是计算机的一些异常事件或其他的内部原因，如除数为 0 中断等。

中断传送方式是 CPU 和外部设备进行输入/输出的有效方式，一直被大多数计算机所采用，它可以避免因反复查询外部设备的状态而浪费时间，从而提高 CPU 的效率。不过，每中断一次，只传送一次数据，数据传送的效率并不高，所以，中断传送方式一般用于低速外设。另外，与查询方式相比，中断传送方式实现比较复杂，对硬件的要求也较高。

9.2.3 直接存储器存取方式

高速外设（例如磁盘机等）由于准备数据的时间短，要求传送的数量较多，同时要求传送的速度快等特点，一般采用直接存储器存取方式，又称为 DMA（Direct Memory Access）方式。在这种方式下高速设备与内存之间直接交换数据。这种方式传送数据是成组进行的。程序控制方式和中断控制方式有一个共同的特点，两种方式的输入/输出操作均以 CPU 为中心，通过执行指令来实现输入/输出操作。因此，必然花费大量 CPU 执行指令周期。对于输入/输出设备与主存储器一次需进行数量较大的信息交换（例如从磁盘读取一个大文件），CPU 的开销就太大了。所以，为提高内存与外设之间交换信息的速度，最有效的办法是不通过 CPU，让信息直接进行传送。即以存储器为中心，在一个专门的控制电路（如 DMA 控制器）的控制下传送信息，不再需要 CPU 的干预，如图 9.9 所示。DMA 方式能使 I/O 设备直接和存储器进行成批数据的快速传送。每个字节一旦到达端口，就直接从接口送到存储器，同样，接口和它的 DMA 控制器也能直接从存储器取出字节并把它送到 I/O 设备中去。

DMA 方式传送数据的一般过程是：先把数据在高速外设中存放的起始位置、数据在内存中存放的起始地址、传送数据长度等参数输出到连接高速外设的接口（DMA 控制器），然后启动高速外设，设备开始准备直接传送数据。当高速外设直接传送准备好后，向处理器发送一个直接传送的请求信号，处理器以最短时间批准进行直接传送，并让出总线控制权，高速外设在其控制器控制下交换数据。数据交换完毕后，由高速外设发出“完成 DMA 中断请求”的信号，并交回总线控制权。处理器响应上述中断，由对应的中断处理程序对高速外设进行控制或对已经传送的数据进行处理。中断返回后，原程序继续运行。

DMA 控制方式对成批数据的传送很方便，也可获得高效率。

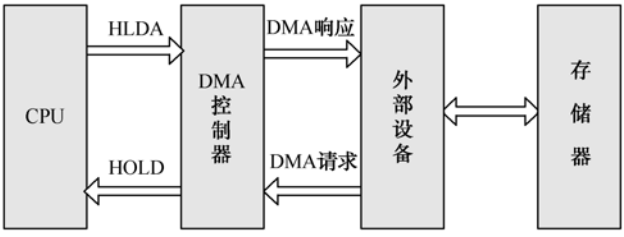


图 9.9 DMA 控制器连接示意图

9.3 中断

中断（interrupt）是现代计算机输入/输出程序设计常用的控制方式。在输入/输出程序中往往需要随时根据设备或环境的需要及时调用某些特定程序去进行某种工作，要求 CPU 中止正在执行的程序而转去处理特殊事件，这样一种操作称为中断。引起中断的事件称为中断源，而处理中断事件的程序称为中断服务程序，程序发生中断的地方称为中断点。

9.3.1　中断的分类

中断程序什么时候被调用是无法预先设计的。8086/8088 CPU 可以处理 256 种类型的中断，这些中断可以来自 CPU 内部，也可以来自 CPU 外部。按引起中断的原因划分，中断源可分为外部中断与内部中断两大类。8086/8088 中断源如图 9.10 所示。

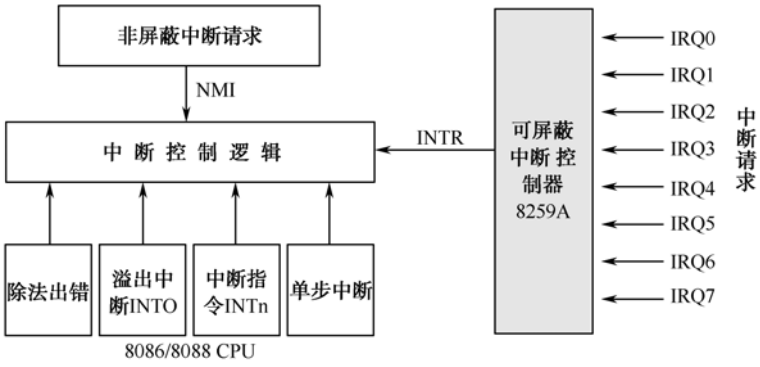


图 9.10　8086/8088 中断源

1. 内部中断

内部中断指由 CPU 内部原因引起的中断。常见的如除法运算出错引起的中断，执行中断指令引起的中断，单步运行时的中断等，它们都是非屏蔽中断。内部中断一般可分为以下四种。

（1）除法出错中断：当除数为“0”或除法中所得商过大时，立即产生一个内部中断。

（2）溢出中断指令 INTO：当执行这条指令时，若由于前面的运算已产生溢出，使 OF=1，这时便产生一个内部中断；若 OF=0，便不产生中断。

（3）中断指令 INT n：操作系统把一些常用的输入、输出、文件处理等基本操作，编制成若干子程序存放在系统内，这些子程序又称为中断服务程序，对它们的调用均用中断指令 INT 来进行，其中 n 为中断服务程序类型号。

（4）单步中断：这是为了调试程序提供的强有力的手段。若单步标志 TF 为 l，则在每条指令执行后，CPU 自动产生一个类型 1 的中断，即单步中断，使程序单步执行，例如，执行 DEBUG 命令时，使用 T 或 P 命令都将置 TF，引起中断。

8086/8088 为每一个中断源都分配一个中断类型码。这些中断类型码有的是系统统一规定的，有的可由用户指定中断类型码。这些中断类型码是各自中断源的唯一编号。统一规定的中断类型码如表 9.2 所示。

表 9.2　中断类型码

中断类型码	中断源	中断类型码	中断源
0	除法出错	8	定时器
1	单步中断	9	键盘
2	非屏蔽性中断（NMI）	B	COM2

续表

中断类型码	中断源	中断类型码	中断源
3	断点中断	C	COM1
4	溢出中断	D	硬盘（并行口）
5	打印屏幕	E	软盘
6	未用	F	打印机

中断类型码 5～31 是系统扩展用的保留部分。表中类型码 3 是用于程序调试的软中断指令的特殊形式，是单字节指令。例如，在 DEBUG 中，设置断点，每当程序运行至断点处，程序就自动停止，等待用户检查程序的运行结果。

由用户自行确定的中断类型码 32～255，可利用可编程的中断控制器 8259A 来设定每一个中断源的中断类型码。

2. 外部中断

由发生在 CPU 外部的某个事件引起的中断称为外部中断。外部中断根据其重要性又可分为非屏蔽中断（NMI）和可屏蔽中断（INTR）。

非屏蔽中断指为应付一些紧急故障而引起的中断，如发生电源掉电、内存或 I/O 总线的奇偶错等异常事件时引起的中断。CPU 不能禁止非屏蔽中断，一旦发生，CPU 一定要找到相应处理程序转入对事件的处理。

可屏蔽中断指可以通过设置而暂时不响应的中断。例如，许多输入/输出设备的中断处理，包括硬盘、软盘、键盘、显示器和各种打印机等引起的一些中断。这些外部设备通过 8259A 可编程中断控制器和 CPU 相连。如果它们产生中断请求，CPU 不一定立即响应，是否响应取决于该外设的中断请求是否被屏蔽，以及 CPU 是否允许响应两个条件。这两个条件分别由 8259A 的中断屏蔽寄存器（IMR）和标志寄存器（Flags）中的中断允许位 IF 控制。

9.3.2 中断向量表

1. 中断向量表

不同的中断源要求的中断处理是不相同的。因此，每一个中断源都设计一个中断处理程序，在 CPU 响应中断后调用对应的中断处理程序。如何找到要调用的中断处理程序的入口地址呢？系统用一张表来保存这些中断处理程序的入口地址，该表就称为中断向量表。中断向量表的每一项保存一个中断处理程序的入口地址，它相当于一个指向中断处理程序的指针，所以就称它为中断向量。中断向量也依次编号为 0～0FFH，n 号中断向量就保存处理中断类型为 n 的中断处理程序的入口地址。所以，一般不再区分中断类型号和中断向量号。

8086/8088 在内存最小地址的 1KB 字节空间建立了一个中断向量表，它确定了中断类型码与该中断源对应的中断处理程序之间的一一对应关系。其中每个中断向量占用 4 个字节，前（低地址）两字节保存中断处理程序入口地址的偏移，后（高地址）两字节保存中断处理程序入口地址的段值，图 9.11 给出了中断向量表的安排，表内存放有 256 个中断处理程序入口地址。按照中断向量表的这种结构和存放位置，根据中断类型号（或向量号）

可方便地计算出中断向量所在单元的地址。设中断向量号为 n，则中断向量所在单元的开始地址是 4*n，那么，

IP←(4*n,4*n+1)

CS←(4*n+2,4*n+3)

即把中断处理程序的入口地址（偏移地址和段地址）分别送入 IP 和 CS 中，就可转到相应的中断服务程序中。

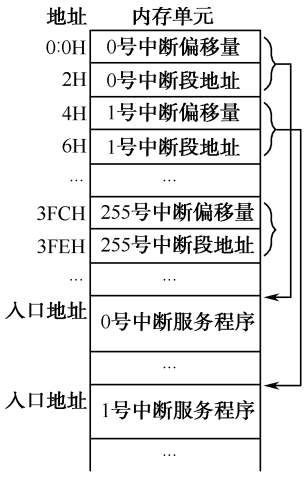


图 9.11　中断向量表

注意：

中断向量不一定非要指向中断处理程序，也可作为指向一组数据的指针，例如，1DH 号中断向量就指向显示器参数，1EH 号中断向量指向软盘基数。当然，如果中断向量 n 没有指向中断处理程序，那么就不应发生类型为 n 的中断。

2. 读取中断向量

下面的程序片段直接从中断向量表中取得 n 号中断向量，并且保存到双字变量 INTRDWCTOR 中。

```
…
XOR     AX, AX
MOV     ES, AX
MOV     AX, ES : [n * 4]
MOV     WORD PTR INTRDWCTOR, AX
MOV     AX,ES : [n * 4+2]
MOV     WORD PTR INTRDWCTOR +2, AX
…
```

与利用 DOS 功能调用设置中断向量一样，实际上一般都利用 DOS 提供的 35H 号系统功能调用取得中断向量。35H 号系统调用的功能是获取中断向量，其出入口参数如下。

入口参数：

AL=中断向量（类型）号

出口参数：

ES=中断处理程序入口地址的段值

BX=中断处理程序入口地址的偏移

下面的程序片段取得 n 号中断向量，并将其保存到双字变量 OLDVECTOR 中。

```
MOV     AH, 35H
MOV     AL, n
INT     21H
MOV     WORD  PTR  INTRDWCTOR+2, ES
MOV     WORD  PTR  INTRDWCTOR, BX
```

3. 设置中断向量

系统程序或应用程序由于某种需要而提供新的中断处理程序时，就要设置对应的中断向量，使其指向新的中断处理程序。将自己编写的子程序入口地址放入对应的中断向量表即可。主要的方法有两种。

（1）直接法

直接法就是利用传送指令，将子程序入口地址直接放入中断向量表中。

例如，已经编写好的中断处理程序如下：

```
INTRF   PROC    FAR
        …
        STI
        IRET
INTRF   ENDP
```

下面的程序片段直接将其设定为中断类型号为 0F1H，代码如下：

```
…
MOV     AX, 0
MOV     ES, AX                          ; 段地址为 0000H
MOV     BX, 0F1H*4                      ; 准备设置 F1H 号中断向量
CLI                                     ; 关中断
MOV     AX, OFFSET INTRF                ; 子程序入口的偏移地址
MOV     ES:[BX], AX                     ; 存入
MOV     AX, SEG INTRF                   ; 子程序入口的段地址
MOV     ES:[BX+2], AX                   ; 存入
STI
…
```

在上面的程序片段中，使用了关中断指令 CLI，目的是保证用于设置中断向量的两条传送指令能够被连续执行。在执行完前一条传送指令后，0F1H 号中断向量就暂时被破坏，既不指向原中断处理程序，也不指向新的中断处理程序。如果此时发生类型为 0F1H 的中断，那么就不能正确地转到中断处理程序执行，这是最糟糕的事。如果能确定当前是处在关中断状态，就不再需要使用关中断指令，也不需要随后的开中断指令。另外，如果能肯定在设置 0F1H 号中断向量的过程中不发生类型为 0F1H 的中断，那么可不考虑是否为关中断状态，这种情况只有在对应的中断处理程序仅供应用程序自己使用时才有可能发生。

（2）利用 DOS 功能调用设置中断向量

实际上，总是尽量避免采用上述直接设置中断向量的方法（有时是必须的），而是通过 DOS 提供的 25H 号系统功能调用来设置中断向量，这样可以避免考虑许多细节。

25H 号系统功能调用是设置中断向量，其入口参数如下：

AL=中断类型号

DS=中断处理程序入口地址的段值

DX=中断处理程序入口地址的偏移

上例可改写如下：

```
…
MOV     DX, OFFSET INTRF                ; 子程序入口的偏移地址
MOV     AX, SEG INTRF                   ; 子程序入口的段地址
MOV     DS, AX
MOV     AH,25H
MOV     AL,0F1H
INT     21H
…
```

注意：

自己设计的中断处理程序所采用的中断类型号一般应为 BIOS 中未用到的，以免出现冲突。一般 60H～66H、F1H～FFH 这些中断类型号是为用户程序所准备的。

9.3.3　中断优先级和中断嵌套

1. 中断优先级

中断优先级是指 CPU 响应中断的优先顺序，规定优先级高的可以中断优先级低的中断服务程序的运行，相反则不可。由于系统中有内部中断、外部中断等多个中断源，当出现多个中断源同时向 CPU 请求中断时，CPU 必须有一个响应次序。一般方法是给各种中断源事先安排一个中断优先级顺序，并用某种方法定义到计算机中。当多个中断源同时申请中断时，CPU 先比较它们的优先级，然后按优先级从高到低的顺序来依次处理各个中断源的中断请求。

8086/8088 每个中断源优先级高低如表 9.3 所示。

表 9.3　中断优先级

优　先　级	中　断　源
高 ↓ 低	内中断（除法出错、INT n、INTO）
	不可屏蔽中断（NMI）
	可屏蔽中断（INTR）
	单步中断

外设的中断请求都通过中断控制器 8259A 传给 CPU 的 INTR 引线。在对 8259A 初始化时规定了 8 个优先级，在正常的优先级方式下，优先级次序由高到低如下：

IRQ0，IRQl，IRQ2，IRQ3，IRQ4，IRQ5，IRQ6，IRQ7

但在必要的情况下，通过设置中断控制器 8259A 中的中断命令寄存器的有关位可改变上述优先级次序。中断命令寄存器的 I/O 端口地址是 20H，其各位的定义如图 9.12 所示。其中的 L2～L0 三位指定 IRQ0～IRQ7 中具有最低优先级的中断请求，R 位和 SL 位控制 IRQ0～IRQ7 的中断优先级的次序。当 R 位和 SL 位全为 0 时，表示采用正常优先级方式，即上述优先级次序。在一般情况下，总采用这种正常优先级方式。

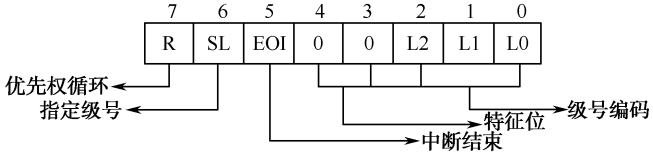


图 9.12　中断命令寄存器

中断命令寄存器中 EOI 位是中断结束位，当把它置为 1 时，表示当前中断处理结束。在对中断控制器 8259A 初始化时规定，在 CPU 响应某个外设的中断请求后，中断控制器 8259A 不再传出中断级相同或较低的外设中断请求，直到 8259A 接收到中断结束命令为止。

例如；CPU 在响应来自 IRQ1 的 9 号键盘中断后，8259A 就不再传出来自 IRQ1～IRQ7 的外设中断请求，直到通知 8259A 键盘中断已结束为止。所以，在外设中断处理程序结束时，要通知 8259A 中断已结束，以便使 8259A 传出中断级相同或较低的外设中断请求，从而使 CPU 响应它们。下面程序片段通知 8259A 当前中断结束。

```
MOV     AL, 20H
OUT     20H, AL
```

在对 8259A 初始化时作上述规定的理由是使 CPU 在响应外设中断请求后，只要开中断，那么就可响应优先级高的外设中断请求，而不会响应优先级相同或低的外设中断请求，待原有中断请求结束后，再去响应它。

注意：

（1）通知中断控制器 8259A 当前中断结束，并非中断返回。只有在执行了中断返回指令后，才返回被中断的程序。

（2）在以后的章节中，如无特别说明，外设中断的优先级次序采用正常优先级次序。

2. 中断嵌套

正在运行的中断处理程序又被其他中断源中断，这种情况称为中断嵌套，如图 9.13 所示。一个正在执行的中断处理程序，在开中断（IF=1）的情况下，能被优先级高于它的中断源所中断。在中断处理过程中，发生内部中断，引起中断嵌套是经常的事。例如，CPU 在执行中断处理程序时，遇到软中断指令，就会引起中断嵌套。

在中断处理过程中，发生非屏蔽中断，也会引起中断嵌套。

由于 CPU 在响应中断的过程中，已自动关中断，所以 CPU 也就不会再自动响应可屏蔽中断。如果需要在中断处理过程的某些时候响应可屏蔽中断，那么可在中断处理程序中安排开中断指令，CPU 在执行开中断指令后，就处于开中断状态，也就可以响应可屏蔽中断了，直到再关中断。所以，如果在中断处理程序中使用了开中断指令，也就可能会发生可屏蔽中断引起的中断嵌套。

8086/8088 没有限制中断嵌套的深度（层次），但会受到堆栈容量的限制。

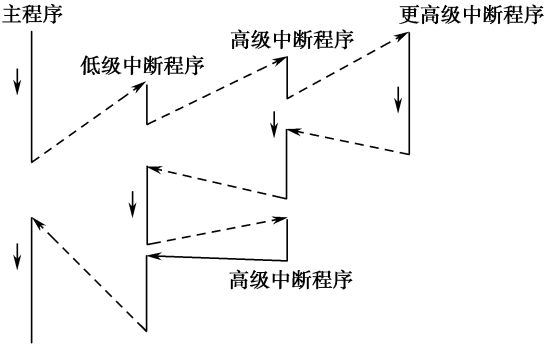


图 9.13　中断嵌套示意图

9.3.4　中断指令

1. 中断指令

若中断指令中立即数为 n，则当执行中断指令后，就产生一个类型号为 n 的中断，从而转入对应的中断处理程序。执行中断指令后的堆栈如图 9.14 所示。中断指令 INT 的一般格式如下。

格式：

```
INT       n
```

其中，n 是一个 0～0FFH 的立即数。

指令执行的操作如下：

（1）将标志寄存器压栈，清除标志位 IF 和 TF。

（2）将现行程序的代码段寄存器 CS 和指令指针 IP 的内容压栈，并把中断处理程序入口地址的低字部分送入 IP 中，高字部分送入 CS 中。

在该指令执行完后，CPU 将转去执行中断处理程序。

例如，为了调用 DOS 系统功能，就在程序中安排如下的中断指令：

```
INT       21H
```

当 CPU 执行该指令后，就产生一个类型为 21H 的中断，从而转入对应的中断处理程序，也即转入 DOS 系统功能服务程序。

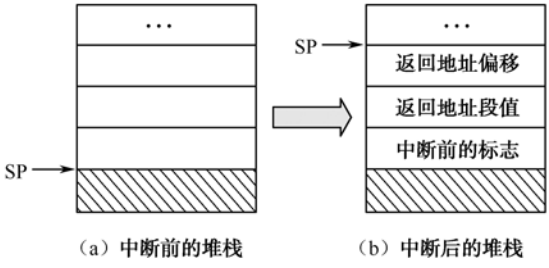


图 9.14　响应中断时的堆栈

2. 溢出指令

8086/8088 提供一条专门检测运算溢出的指令，当溢出标志位 OF 为 1 时，引起中断。如果溢出标志位 OF 为 0，则执行该指令后并不产生溢出中断。指令格式如下。

格式：

```
INTO
```

该指令影响标志位 IF 和 TF，溢出中断的类型号规定为 4。

3. 中断返回指令

当一个中断处理程序执行完毕后，CPU 将恢复被中断的现场，返回到调用中断指令的下一条指令处。为了实现此项功能，指令系统提供了一条专用的中断返回指令。该指令的格式如下。

格式：

```
IRET
```

该指令执行的过程基本上是 INT 执行的逆过程，具体操作如下：

（1）从栈顶弹出内容送入 IP 中。

（2）从新栈顶弹出内容送入 CS 中。

（3）从新栈顶弹出内容送入标志寄存器中。

9.3.5 中断过程

中断过程类似于子程序的调用过程，每一个完整的中断过程通常要经历中断请求、中断优先级判定、中断响应、中断处理及中断返回等 5 个步骤。

1. 中断请求

中断请求是由来自外设的 I/O 请求、计算机的一些异常事故或其他内部原因而引起的。中断源产生中断请求的条件对不同的中断源是不一样的。如除法出错中断请求的条件是执行除法指令时，除数为 0 或商超出范围；对于输出设备，则是在从端口取走一个字符后产生中断请求；对于输入设备，则是输入一个字符后提出中断请求。中断请求通过连线与 8086/8088 的中断逻辑相连，等待 CPU 响应并处理所请求的中断。

2. 中断优先级判定

按规定的优先级次序对各中断源进行优先级判定，通常由硬件完成。其中，可屏蔽中断的优先级分为 8 级，默认情况下的优先级次序为 IR0、IRl、IR2、IR3、IR4、IR5、IR6、IR7。

3. 中断响应

CPU 在每执行完一条指令后，均要查询是否有中断请求。当有中断请求时，就按照图 9.15 所示的顺序给予响应或不响应。当响应中断后，转入中断响应周期，系统硬件自动完成下列动作。

（1）取中断类型号 n，主要针对来自硬件的可屏蔽中断请求。此时，CPU 会通知中断源，将其中断类型号放在数据线上，然后由 CPU 读入。但对于内部中断（因为在指令中已含有中断类型号），以及来自硬件的非屏蔽中断请求（因为其中断类型号固定），则不需要这一步。

（2）标志寄存器（PSW）内容入栈。

（3）当前代码寄存器（CS）内容入栈。

（4）当前指令计数器（IP）内容入栈。

（5）禁止外部中断和单步中断（IF←0，TF←0）。

（6）从中断表中 4*n 处取一个字的内容送入 IP 中，4*n+2 处取一个字的内容送入 CS 中。

（7）转中断处理程序。

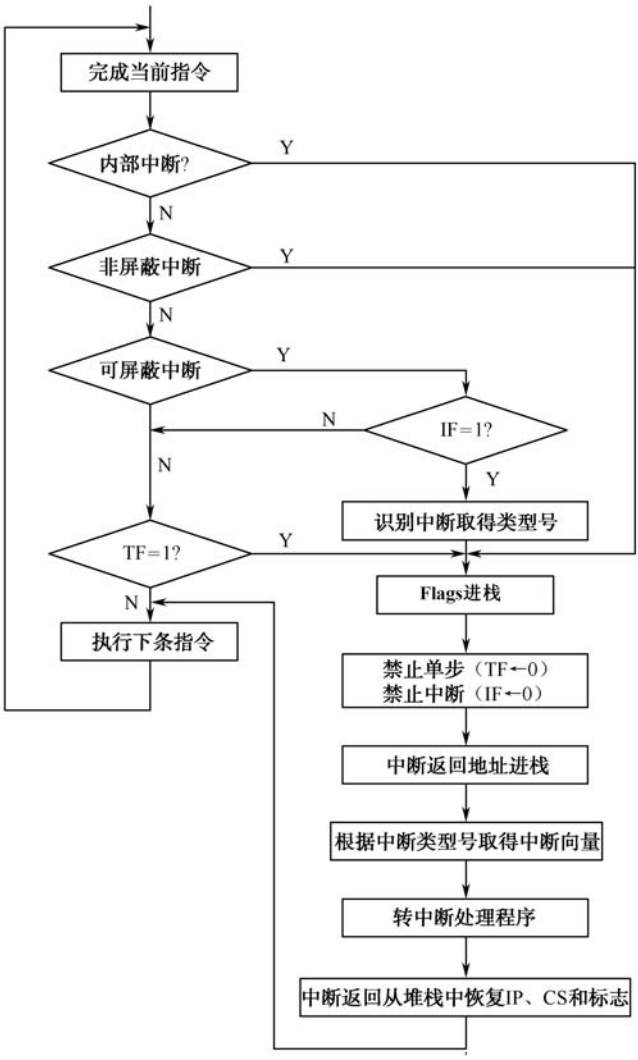


图 9.15　中断处理过程

4. 中断处理

中断响应后，进入中断处理阶段。中断处理是由中断服务子程序完成的，与此同时，由硬件电路关闭 CPU 内的允许中断触发器，禁止新的可屏蔽中断（前面未提到它，实际上是存在的）。中断服务程序一般由 3 个部分组成：起始部分（也称为前处理部分）、主体部分和结尾部分（也称为后处理部分）。中断处理是由硬件和软件共同完成的。

（1）起始部分

起始部分主要是保护现场，开放（允许）中断，将 CPU 内部寄存器的内容依次压入堆栈，同时开放（允许）中断（IF=1）。在中断响应时，CPU 内的中断允许触发器自动关闭。其目的是在替换新老屏蔽字和保护现场操作时，禁止一切中断；否则，会引起 CPU 现场混乱。现场保护以后，就应开放中断，以便在本次中断处理过程中可以响应更高级的中断请求。若本次中断过程不允许再被中断，就可以不进行这一操作。

（2）主体部分

主体部分是中断本身的处理程序，起始部分完成后，接着进入中断处理程序的主体部

分，即真正的服务程序，完成相应的操作。这可根据不同的任务设计相应的服务程序段。有的是进行数据传送，有的是设备检查，有的是数据传送完毕后的结束处理。根据不同情况，主体部分可以是一条指令，也可以是一段程序。

（3）结尾部分

结尾部分是恢复现场。首先关闭（禁止）中断（IF=0），防止恢复现场过程被其他中断所打扰。禁止中断是通过关闭中断指令来实现的。然后恢复本次中断的现场，把保护现场中堆栈的内容弹回各个寄存器。其次清除中断请求寄存器相应位，表明本次中断服务程序已经结束。最后开中断（IF=1）。在中断服务程序的最后要开中断，以便 CPU 在返回主程序后能响应新的中断。

中断处理程序的基本形式如下：

```
INTERRUP  PROC   FAR
          PUSH   AX
          PUSH   BX
          PUSH   CX
          PUSH   DX
          PUSH   SI          ; 开始部分，现场保护
          PUSH   DI
          PUSH   BP
          PUSH   DS
          PUSH   ES
          …                  ; 主体部分，中断处理
          POP    ES
          POP    DS
          POP    BP
          POP    DI
          POP    SI          ; 结尾部分，现场恢复
          POP    DX
          POP    CX
          POP    BX
          POP    AX
          IRET               ; 中断返回
INTERRUP  ENDP
```

5. 中断返回（IRET）

每个中断处理程序的最后，都要使用中断返回指令 IRET。在中断处理程序的最后，先用指令清除本次中断 EOI 标志，表示本次中断处理完毕，系统可以接受其他中断。接着用 IRET 指令依次实现 IP 内容的出栈、CS 内容的出栈和状态寄存器的内容出栈，继续执行被中断的程序。这两条指令都是在执行完后，再执行一条指令才能响应新的中断，所以

保证了一个中断处理的完整结束。

中断和子程序调用之间的工作过程非常相似，都是暂停当前程序的执行，转而执行另一程序段，当该程序段执行完时，CPU 都自动恢复原程序的执行。但两者却有本质的差别，其差别如下。

（1）子程序调用一定是程序员在编写源程序时事先安排好的，是可知的，而中断是由中断源根据自身的需要产生的，是不可预见的（用指令 INT 引起的中断除外）。

（2）子程序调用是用 CALL 指令来实现的，但没有调用中断的指令，只有发出中断请求的事件（指令 INT 是发出内部中断信号，而不要理解为调用中断处理程序）；

（3）子程序的返回指令是 RET，而中断处理程序的返回指令是 IRET。

（4）在通常情况下，子程序是由应用系统的开发者编写的，而中断处理程序是由系统软件设计者编写的。

所以中断的处理要比“转子程序”指令的执行复杂，中断处理程序与子程序的功能和编写也不同。一般子程序是在计算机程序中，能够完成一定功能的一串指令，它可以成为主程序的一部分，也可以在主程序中的不同地方使用。而中断处理程序的主要功能是完成某种外设与主机之间的信息传送，每种外部设备的传送都有自己专用的中断处理程序。中断处理程序的编写结构有特殊的要求。它的前处理部分要有保存现场、交换屏蔽字和开放中断的功能，后处理部分应有关闭中断、恢复现场、恢复屏蔽字和开放中断等功能，且在前处理部分和后处理部分是不允许有其他高级中断源发出中断的。而子程序在编写上没有一定的格式规范。

9.3.6　中断处理程序的设计

中断处理程序的功能各式各样，但是除去所处理的特殊功能外，所有的中断处理程序都有相同的结构模式。大部分中断处理程序由 BIOS 或 DOS 系统提供。程序员编写自己需要的中断处理程序时，应注意下面几点。

（1）程序中不用数据段，一般将数据变量直接放入代码段中。

（2）程序中不用堆栈段，中断处理程序借用调用者的堆栈段。

（3）在进入中断处理程序后，首先要保护现场。在程序中用到的寄存器要压入堆栈保护起来，在程序结束前恢复各寄存器内容，使用时要注意堆栈先进后出的特性。

（4）在中断处理程序中，若允许中断嵌套，应在保护现场后，用 STI 指令开中断，在中断处理程序的工作结束后，在恢复现场返回之前用 CLI 指令关中断。

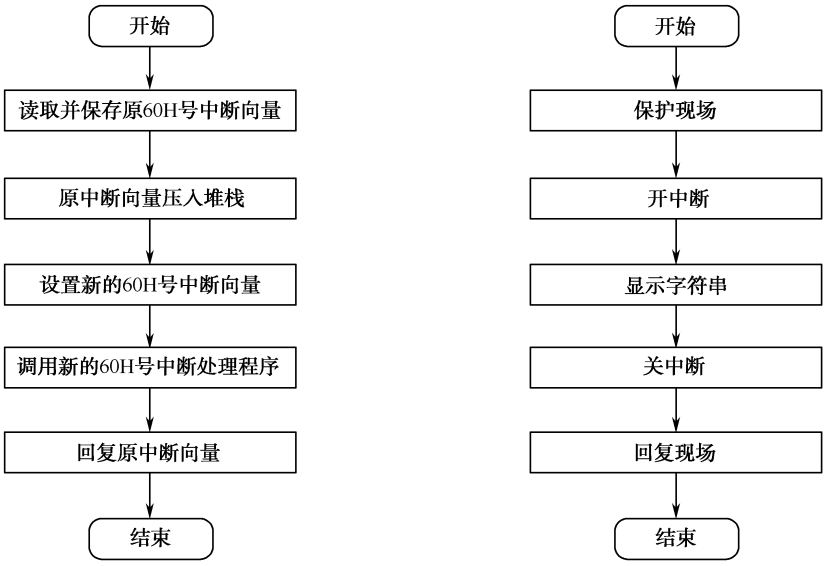
【例 9.2】　编写一个中断处理程序，中断类型号为 60H，功能为显示一个字符串“I Love China”。

（1）分析问题

此程序需要完成设置中断向量的任务，并在中断处理程序中，调用 INT 21H 显示字符串。为了避免破坏原 60H 号中断向量，首先读取并保存原中断向量，以便在测试新的 60H 号中断及其处理程序后，再恢复原中断向量。作为中断程序，程序中不用数据段，将待显示的字符串直接放在代码段中。

（2）流程图

流程图如图 9.16 所示。



（a）主程序　　（b）中断处理程序

图 9.16　程序流程图

（3）编写程序

程序如下：

```
CODE    SEGMENT
        ASSUME  CS:CODE, DS:CODE
MAIN    PROC    FAR
START:  MOV     AL,60H                          ; 获取原 60H 中断向量
        MOV     AH,35H
        INT     21H                             ; 读取的中断向量位于 ES:BX 中
        PUSH    ES                              ; 压入堆栈保存
        PUSH    BX                              ; 设置新的 60H 中断向量
        MOV     AX,SEG      INT_PROGRAM         ; 中断向量的段地址放在 DS 中
        MOV     DS,AX
        MOV     DX,OFFSET   INT_PROGRAM         ; 中断向量的偏移地址放在 DX 中
        MOV     AL, 60H
        MOV     AH, 25H                         ; 用 INT 21H 的 25H 号功能设置中断向量
        INT     21H                             ; 调用新的 60H 号中断进行测试
        INT     60H                             ; 恢复原 60H 中断向量
        POP     DX                              ; 把原中断向量弹出堆栈
        POP     DS
        MOV     AL,60H
        MOV     AH,25H                          ; 恢复原中断向量
        INT     21H
        MOV     AX,4C00H                        ; 返回 DOS
```

```
            INT     21H
MAIN        ENDP
SHOW        DB      ' I Love China '                ; 待显示字符串
INT_PROGRAM         PROC    FAR                     ; 新 60H 中断处理程序
            PUSH    AX                              ; 保护现场
            PUSH    DX
            PUSH    DS
            STI                                     ; 开中断
            PUSH    CS                              ; 设置 DS-CS
            POP     DS
            MOV     DX, OFFSET  SHOW
            MOV     AH,9
            INT     21H                             ; 显示字符串
            CLI                                     ; 关中断
            POP     DS
            POP     DX
            POP     AX
            IRET
INT_PROGRAM         ENDP
CODE        ENDS
            END     START
```

9.4　BIOS 与 DOS 中断调用

在实际应用中，主机与外部设备的数据传送大部分是采用中断方式来实现的，固化在主机系统主板上较高地址的 ROM 中的基本输入/输出系统 BIOS（Basic Input/Output System）提供了系统加电自检、引导并装入操作系统、主要 I/O 设备的中断处理等功能程序。磁盘操作系统 DOS（Disk Operating System）是重要的操作系统，是由 BIOS 在开机后自动装入内存的。

9.4.1　BIOS 与 DOS 中断概述

BIOS 包含了主要 I/O 设备的处理程序和许多常用例行程序。它们一般以中断处理程序的形式存在，不仅处理系统的全部中断，还提供对主要 I/O 接口的控制功能，如键盘、显示器、磁盘、打印、日期和时间等。BIOS 是模块化的结构形式，每个功能模块的入口地址都在中断向量表中。在 1KB 中断向量表中，共有 256（00H～FFH）个中断调用的入口地址。对这些中断调用是通过软中断 INT 实现的。软中断指令的操作数就是中断类型码。这样，用户就可以在不必了解中断处理程序的内部结构的基础上使用中断处理程序。

DOS 建立在 BIOS 的基础上，通过 BIOS 操纵控制硬件。例如，DOS 调用 BIOS 显示 I/O 程序完成显示输出，调用打印 I/O 程序完成打印输出，调用键盘 I/O 程序完成键盘输入。尽管 DOS 和 BIOS 都提供某些相同的功能，但它们之间的层次关系是明显的。用户应用程序、DOS 系统、BIOS 系统和外部设备之间的关系如图 9.17 所示。

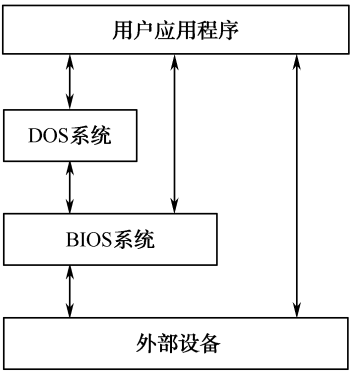


图 9.17　各类系统中断间层次关系

在用户应用程序中，若直接通过端口来操作硬件或外设，那么，其处理过程没有额外的多余工作，处理速度显然是最快的，但这样做，无疑使用户程序具有了很大的局限性。硬件环境的改变将直接影响程序的正常运行。若用户程序通过调用 DOS 功能来实现其所需功能，那么，应用程序与低层硬件相距较远，操作最终的对象需要经过中间环节，处理速度肯定受到一定的损失。但这种应用程序适应性强，应用范围广，对硬件的依赖性最小。

由于 BIOS 介于 DOS 和具体硬件之间，所以，调用 BIOS 的功能是一个很好的折中方案。程序员可在以下三种情况下考虑使用 BIOS 的功能。

（1）BIOS 提供的功能，而 DOS 没有提供该功能的情况；

（2）不能利用 DOS 功能调用的情况（可能因为某些具体应用的限制）；

（3）基于处理速度的考虑，需要绕过 DOS 层的情况。

综上所述，可以归纳出如下结论：使用中断的层次越高，它与硬件设备相关程度就越低，处理速度也就越低，但用户程序的适用范围较广；反之亦然。常用的 BIOS 中断类型和常用的 DOS 中断类型如表 9.4 和表 9.5 所示。

表 9.4　常用的 BIOS 中断类型

中断类型号	功　能	中断类型号	功　能
0H	除法出错	10H	显示器
1H	单步中断	11H	设备检验
2H	非屏蔽中断	12H	内存大小
3H	断点	13H	磁盘
4H	溢出	14H	通信
5H	打印屏幕	15H	I/O 系统扩充
6H	保留	16H	键盘
7H	保留	17H	打印机
8H	8254 系统定时器	18H	驻留 BASIC
9H	键盘	19H	引导
0AH	保留	1AH	时钟
0BH	保留（通信）	1BH	键盘 Break
0CH	保留（通信）	1CH	定时器
0DH	保留（Alt 打印机）	1DH	显示参量
0EH	软盘	1EH	软盘参量
0FH	打印机	1FH	图形字符扩充

表 9.5　常用的 DOS 中断类型

中断类型号	功　能	中断类型号	功　能
20H	程序终止	27H	终止并驻留程序
21H	功能调用	28H	键盘忙循环
22H	结束地址	29H	快速写字符
23H	Ctrl_Break 出错地址	2AH	网络接口
24H	严重出错处理	2EH	执行命令
25H	绝对磁盘读	2FH	多路转接接口
26H	绝对磁盘写	30H～3FH	保留给 DOS

在 DOS 功能调用中，主要使用 21H 中断，因为在该中断中包括有常用的键盘、显示器、打印机、磁盘、时钟等功能。DOS 功能调用的子程序已按顺序编号（功能号 00H～6CH），其调用的方法是：

（1）功能号→AH。

（2）入口参数→指定寄存器。

（3）INT 21H。

用户只要给出以上三方面信息，DOS 就可根据所给信息自动转入相关的中断处理子程序执行。

在 BIOS 中断类型中，0H～9H 号中断为硬中断，一般不用程序来调用，是由 I/O 硬件事件触发的，其他中断就可以由用户通过指令来调用。其主要的 I/O 设备有键盘、打印机、显示器、时钟、磁盘和异步通信端口等。用户在汇编语言程序设计中可使用软中断指令 INT n 调用 BIOS 程序，其中 n 是中断类型码，如用指令 INT 16H 可调用键盘驱动程序。用不同的功能号加以区分，并约定功能号存放在寄存器 AH 中，其调用方法与 DOS 功能调用类似。其调用的方法是：

（1）功能号→AH。

（2）入口参数→指定寄存器。

（3）指令 INT n 实现对 BIOS 子程序的调用。

【例 9.3】　分别用 BIOS 功能调用和 DOS 功能调用通过打印机输出一个字符'B'。

（1）BIOS 功能调用下的字符输出程序代码为：

```
MOV     AH, 0          ; 0 号功能
MOV     DX, 0          ; 0 号打印机
MOV     AL, 'B'        ; 要打印的字符送到 AL 中
INT     17H            ; BIOS 的 17H 中断
```

（2）DOS 功能调用下的字符输出程序代码为：

```
MOV     DL, 'B'        ; 要打印的字符送 DL 中
MOV     AH, 05H        ; 5 号功能调用
INT     21H            ; DOS 的 21H 中断
```

BIOS 中断更靠近机器，但是程序较复杂，DOS 中断使用较方便，BIOS 中断功能要多一些，用户在使用时可以根据自己的需要来合理选择。

9.4.2 键盘 BIOS 功能调用

键盘是计算机最基本的一种输入设备，用来输入信息，以达到人机对话的目的。键盘主要由 3 种基本类型的键组成。

（1）字符数据键。

（2）扩展功能键，如 Home、End、BackSpace、Delete、Insert、Pageup、PageDown，以及功能键 Fl～Fl0。

（3）和其他键组合使用的控制键，如 Alt、Ctrl 和 Shift 等。

键盘和主机通过 5 芯电缆相连，这 5 根线分别是电源线、地线、复位线，以及键盘数据线和键盘时钟线。PC 系列的键盘触点电路按 16 行×8 列的矩阵来排列，用单片机 Intel 8048 来控制对键盘的扫描。按键的识别采用行列扫描法，即根据对行线和列线的扫描结果来确定闭合键的位置，这个位置值称为按键的扫描码，通过数据线将 8 位扫描码送往主机。当在键盘上“按下”或“放开”一个键时，如果键盘中断是允许的（21H 端口的第一位等于 0），就会产生一个类型 9 的中断，就会转入 9 号中断处理程序。把 9 号中断处理程序称为键盘中断处理程序，它属于外设中断处理程序这一类。

该处理程序从 8255 可编程外围接口芯片的输入端口读取一个字节，这个字节的低 7 位是按键的扫描码。最高位为 0 或者为 1，分别表示键是“按下”状态或“放开”状态。按下时，取得的字节称为“通码”；放开时，取得的字节称为“断码”。如 Esc 键按下，取得的通码为 01H（00000001B）；放开 Esc 键时，会产生一个断码 81H（10000001B）。

BIOS 键盘处理程序将取得的扫描码转换成相应的字符码，大多数的字符码是一个标准的 ASCII 码。没有相应 ASCII 码的键，如 Alt 和功能键（Fl～Fl0），字符码为 0；还有一些非 ASCII 码键也产生一个指定的操作。键盘缓冲区是一个先进先出的环形队列，其所占内存区域如下：

```
BUFF_HEAD   DW      ?           ; 其内存地址为 0000:041AH，缓冲区头指针
BUFF_TAIL   DW      ?           ; 其内存地址为 0000:041CH，缓冲区尾指针
KB_BUFF     DW      16 DUP(?) ; 其内存地址为 0000:041EH，该缓冲区的默认长度为 16 个字
```

虽然缓冲区的本身长度为 16 个字，但因键盘缓冲区是一个环形队列，出于判断“队列满”的考虑，它最多只能保存 15 个键盘信息。当缓冲区满时，系统将不再接受按键信息，而会发出“嘟”的声音，以示要暂缓按键。当 BUFF_HEAD=BUFF_TAIL 时，表示无键盘输入。

BIOS 提供的键盘中断类型号为 16H，它的中断处理程序又分为 3 个功能，通过 AH 中的功能号 0～2 来选择。BIOS 键盘功能调用如表 9.6 所示。

表 9.6 键盘 BIOS 功能调用 INT 16H

AH	功 能	输入参数	返回结果
0	从键盘输入一个字符	AH=0	AH=键盘扫描码，AL=字符的 ASCII 码或 AL=0
1	读键盘缓冲区的字符	AH=1	ZF=0 时，AH=扫描码，AL=字符的 ASCII 码或 0 ZF=1 时，表示无键按下，键盘缓冲区为空
2	读键盘状态字节	AH=2	AL=键盘状态字节

1. 0 号功能

功能：从键盘读取字符到 AL 寄存器中。

输入参数：　AH=0。

返回结果：　AH 为输入字符的扫描码；

　　AL 为输入字符的 ASCII 码或者 AL 为 0。

调用方法为：

```
MOV     AH, 0
INT     16H
```

【例 9.4】　编制程序段，从键盘读一个字符，把扫描码存到 KEY1 单元，ASCII 码存到 KEY2 单元。

程序段如下：

```
KEY1    DB      ?
KEY2    DB      ?
        MOV     AH, 0
        INT     16H
        MOV     KEY1, AH
        MOV     KEY2, AL
```

2. 1 号功能

功能：读键盘缓冲区字符到 AL 寄存器中，当有键按下时，ZF=0，否则 ZF=1。

输入参数：　AH=1。

返回结果：　当 ZF=0 时，AH 为扫描码，AL 为字符的 ASCII 码或 0；

　　当 ZF=1 时，表示无键按下，键盘缓冲区为空。

如果有键按下，1 号功能的作用如同 0 号功能；如果无键按下，则直接返回。所以，1 号功能读键盘是探测读。

【例 9.5】　在一个程序中，加入程序段，检测是否按下 Esc 键，如果按下了，则退出程序；否则，继续执行程序。

程序段如下：

```
        MOV     AH, 1           ; 1 号功能
        INT     16H             ; BIOS 的读键盘缓冲区的字符
        JZ      CONT            ; ZF=1，无键按下继续执行程序
        CMP     AH, 0IH         ; ZF=0，有键按下，判断是否为 Esc 键的扫描码
        JZ      EXIT            ; 是 Esc 键的扫描码退出程序
CONT:   …                       ; 继续执行程序
EXIT:   …                       ; 退出程序
```

3. 2 号功能

功能：读取键盘上特殊功能键的状态。

输入参数：　AH =2。

返回结果：　AL 中为特定功能键的状态。

键盘上有一些控制键和双态键，如 Shift、Ctrl、Alt、Num Lock 和 Caps Lock 等，当按下这些键时，键盘处理程序并不返回扫描码，而是设置键盘状态字节的相应位。键盘状态字节如图 9.18 所示，其中高 4 位记录双态键的变化情况，低 4 位反映控制键是否被按下。

用 INT 16H 的 2 号功能可以读取此状态字节。键盘状态字节各位定义如表 9.7 所示。

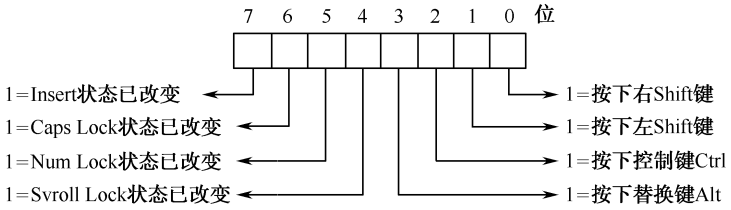


图 9.18　键盘状态字节

表 9.7　键盘状态字节各位定义

键盘状态字节（AL）	表示的意义	键盘状态字节（AL）	表示的意义
80H	Insert 插入键有效。	08H	Alt 键按下状态
40H	大写状态有效(Caps Lock)	04H	Ctrl 控制键按下状态
20H	为数字状态(Num Lock)	02H	为左边上档键有效(Left+Shift)
10H	为停止屏幕滚动状态(Scroll Lock)	01H	为右边上档键有效(Right+Shift)

调用方法为：

```
MOV AH, 2
INT  16H
```

【例 9.6】　编写一个程序，要求完成如下功能：读键盘，并把所按键显示出来，在检测到按下 Shift 键后，就结束运行。

（1）分析问题

使用 INT 16H 的 2 号功能取得变换键状态字节，与 00000011B 进行与操作，进而判断是否按下了 Shift 键。在调用 0 号功能读键盘之前，先调用 1 号功能判键盘是否有键可读；否则会导致不能及时检测到用户按下的 Shift。

（2）流程图

流程图如图 9.19 所示。

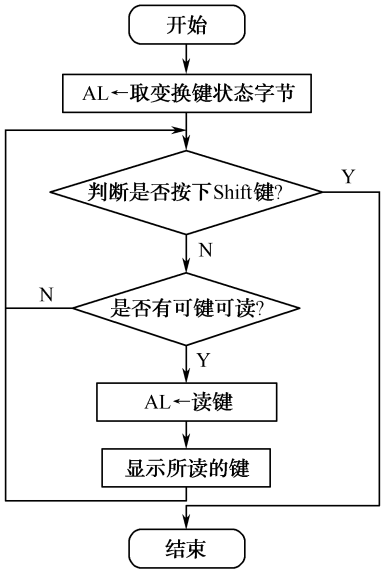


图 9.19　读键盘按键的程序流程图

（3）编写程序

程序如下：

```
DATA      SEGMENT
LR_SHIFT  DB      00000011B                ; 键盘左右侧 Shift 键状态相加
DATA      ENDS
CODE      SEGMENT
          ASSUME  CS: CODE, DS, DATA
START:    MOV     AX, DATA
          MOV     DS, AX
AGAIN:    MOV     AH, 2                    ; 取变换键状态字节
          INT     16H
          TEST    AL, LR_SHIFT             ; 判是否按下 SHIFT 键
          JNZ     OVER                     ; 按下，转到 OVER
          MOV     AH, 1
          INT     16H                      ; 是否有键可读
          JZ      AGAIN                    ; 没有，转到 AGAIN
          MOV     AH, 0                    ; 读键
          INT     16H
          MOV     DL, AL                   ; 显示所读键
          MOV     AH, 6
          INT     21H
          JMP     AGAIN                    ; 继续
OVER:     MOV     AH, 4CH
          INT     21H                      ; 结束
CODE      ENDS
          END     START
```

9.4.3　键盘 DOS 功能调用

前面介绍了键盘的 BIOS 功能调用，而在实际应用中，对于一般的键盘操作，最好不用对硬件依赖大，又不方便的 BIOS 功能调用，而尽量使用对硬件依赖小，又很方便的 DOS 功能调用。表 9.8 中列出了 DOS 功能调用 INT 21H 中与键盘输入有关的功能调用。下面介绍 DOS 键盘操作。

表 9.8　键盘 DOS 功能调用 INT 21H

AH	功　能	输入参数	返回结果
01H	从键盘输入一个字符并回显		AL=字符
06H	测试读取键盘的字符	DL=0FFH	ZF=0，有输入，AL=字符 ZF=1，无输入，AL=0
07H	从键盘输入一个字符，但不回显		AL=字符
08H	从键盘输入一个字符，不回显，检查 Ctrl+Break		AL=字符
0AH	输入字符到缓冲区	DS:DX 为缓冲区首地址	字符及计数已填好
0BH	读取键盘输入状态		AL=0FFH(有输入)
0CH	清键盘缓冲区，并调用一种键盘功能	AL=功能号(1，6，7，8，A)	

1. 01H 号功能调用

功能：从键盘读入单个字符，将字符回显在屏幕上，同时将输入字符送入 AL 寄存器中，如果读到的字符是 Ctrl+Break 键，则结束程序执行。

输入参数：AH=01H。

返回结果：AL 为输入字符的 ASCII 码。

调用格式：

```
MOV     AH, 01H
INT     21H
```

2. 06H 号功能调用

功能：直接读写控制台。

输入参数：AH=06H；若 DL=0FFH，表示输入键盘字符；若 DL=0～0FEH，表示输出字符。

返回结果：输入时，当 ZF=0，表示有字符输入，AL 为输入字符的 ASCII 码；
当 ZF=1，表示无字符输入，AL=0。
输出时，无返回结果。

调用格式：

```
MOV     AH, 06H
MOV     DL, 0FFH
INT     21H
```

3. 07H 号功能调用

功能：从键盘读入一个字符，不回显在屏幕，同时将输入字符送到 AL 寄存器中，也不检测是否为“Ctrl+Break”键。

输入参数：AH=07H。

返回结果：AL 为输入字符的 ASCII 码。

4. 08H 号功能调用

功能：与 1 号功能基本相同，只是字符不回显在屏幕。

输入参数：AH=08H。

返回结果：AL 为输入字符的 ASCII 码。

5. 0AH 号功能

功能：从键盘读入一个字符串到内存缓冲区。

输入参数：AH=0AH，DS:DX 为内存缓冲区首地址。

返回结果：输入字符串及其个数填入内存缓冲区中。

INT 21H 的 0AH 号功能调用可以从键盘读入一个字符串，存入用户定义的缓冲区中。使用这一功能，首先要在程序的数据段定义一个缓冲区，缓冲区的第一个字节由用户预置为缓冲区的最大容量，即允许输入的最大字符数。如果输入的字符数超过了此数，则超出的字符会被丢弃，同时响铃。缓冲区的第二个字节开始存放实际输入的字符串，字符串以回车结束。

调用格式：

```
MOV     DX, OFFSET   MAXLEN                    ; 缓冲取首地址
MOV     AH, 0AH
```

```
INT     21H
```

6. 0BH 号功能

功能：INT 21H 的 0BH 号功能可以检测是否有键按下。它并不返回按键的 ASCII 码或扫描码，通过这一功能，在程序运行的同时，可以检测键盘的状态，看是否按下了任意键来终止程序或者退出循环。

7. 0CH 号功能

INT 21H 的 0CH 号功能可以清除键盘缓冲区，并执行 AL 中指定的一种键盘功能调用，使用 0CH 功能可以使程序在输入一个字符前，将以前输入的字符清除掉，避免由于偶然超前输入的字符而出现错误。

【例 9.7】 编写一个程序，要求功能为：输入一个字符串到内存缓冲区，最后输出缓冲区内容。

（1）分析问题

使用 INT 21H 的 0AH 号功能读取字符串存放在缓冲区中。再使用 INT 21H 的 2 号功能循环依次读取字符并显示出来。

（2）流程图

流程图如图 9.20 所示。

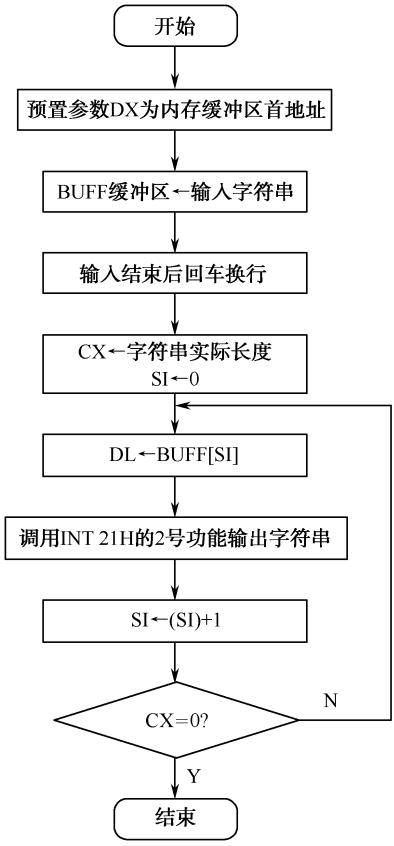


图 9.20　读键盘按键的程序流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
MAXLEN DB       100                             ; 定义输入字符串的最大长度
ACTLEN  DB      ?                               ; 存放输入字符串的实际长度
BUFF    DB      100 DUP(?)                      ; 存放输入的字符串
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS: CODE, DS: DATA
START:
        MOV     AX, DATA
        MOV     DS, AX
        MOV     DX, OFFSET    MAXLEN            ; 预置参数 DX 为内存缓冲区首地址
        MOV     AH, 0AH
        INT     21H                             ; 调用 DOS 功能，输入字符串
        MOV     AH, 2
        MOV     DL, 13
        INT     21H                             ; 输出回车
        MOV     AH, 2
        MOV     DL, 10
        INT     21H                             ; 输出换行
        MOV     CL, ACTLEN
        MOV     CH, 0                           ; 将字符串的实际长度送 CX 寄存器
        MOV     SI, 0
LOOP1:  MOV     DL, BUFF[SI]
        MOV     AH, 2
        INT     21H                             ; 调用 2 号功能输出字符串
        INC     SI                              ; SI 加 1
        LOOP    LOOP1
        MOV     AH, 4CH
        INT     21H
CODE    ENDS
        END     START
```

说明：

在使用 INT 21H 的 0AH 功能输入字符串时，回车键结束字符串输入。

9.4.4 显示器 BIOS 功能调用

1. 显示模式

显示器是一个重要的输出设备，它通过显示卡与计算机系统相连。显示器的显示屏通常称之为屏幕。计算机系统中的显示器都有两种显示方式：文本显示方式和图形显示方式。在 DOS 操作系统环境下，其默认的显示方式为文本显示方式，而在 Windows 操作系统环境下，其显示方式是图形显示方式，其绝大多数操作界面是以图形界面的窗口形式出现的。

（1）文本显示方式

文本显示方式是指以字符为最小单位的显示方式，每个字符都是以矩形块形式显示的。在 BIOS ROM 中存有多种不同大小矩阵的字符集，主要的显示字符集矩阵为：8×8（标准）、8×14 和 8×16 等。

在常用的文本显示模式（模式 3）下，屏幕被划分成 25 行，每行可显示 80 个字符，所以，每屏最多可显示 2000（80×25）个字符。为了便于标识屏幕上的每个显示位置，就用其所在行和列来表示之，并规定：屏幕的左上角坐标为（0, 0），右下角坐标为（24, 79）。

在显示字符时，用一个字节存储该字符的 ASCII 码，用另一个字节存储其显示属性，即显示颜色和方式。彩色显示器的字符显示属性定义如图 9.21 所示，存储内容与显示位置的对应关系如图 9.22 所示。在 80×25 的文本显示模式下，满屏可显示 2000 个字符，也就需要 4000 个字节来存储一屏的显示信息。

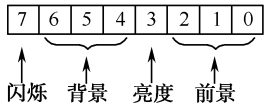


图 9.21　字符显示属性定义

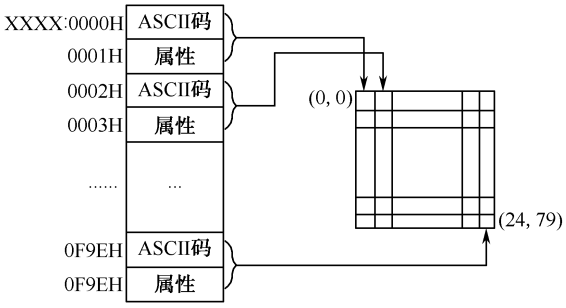


图 9.22　存储内容与显示位置的对应关系示意图

（2）图形显示方式

在该显示方式下，可以看到优美的图像和 VCD，浏览丰富多彩的网页。图形显示的最小单位是像素，对每个像素可用不同的颜色来显示。所以，在显示缓冲区内记录的信息是屏幕各像素的显示颜色。

由于各种图形显示模式所能显示的颜色和像素是不同的，它决定了显示缓冲区的存储方式也是不同的。下面给出三个具体的图形显示模式及其存储方式，通过它们可看出各种显示模式在显示缓冲区存储方式上的明显差异。

① 4 色 320×200 图形显示模式。由于每个像素只能是四种颜色之一，而四种情况用 2 位二进制就可表示，所以，一个字节可表示 4 个像素的显示颜色，存储一行上的所有像素信息就需要 80 个字节。

在具体存储过程中，它又把偶数行像素和奇数行分开来存储。偶数行和奇数行的像素总数各有 32000 个，也都需要 8000 个字节来存储，并规定：偶数行像素从 0B800:0000H 开始存储，奇数行像素从 0B800:2000H 开始存储。该显示模式的存储形式如图 9.23 所示。

② 16 色 640×480 图形显示模式。640×480 图形显示模式共有 307200 个像素，每个像素可选用 16 种颜色，它需要用 4 位二进制来表示。该显示模式在存储显示信息时，把该 4 位分在四个位平面 P1、P2、P3 和 P4 上，所以，位平面 Pi（i=1,2,3,4）共有 307200 个二

进制位，即有 38400 个字节。其显示缓冲区的存储形式如图 9.24 所示。若要改变某个像素的显示颜色，则需要在每个位平面上修改其对应的二进制位。所以，若用操作显示缓冲区的方法来改变像素的颜色，那么编程将困难得多。

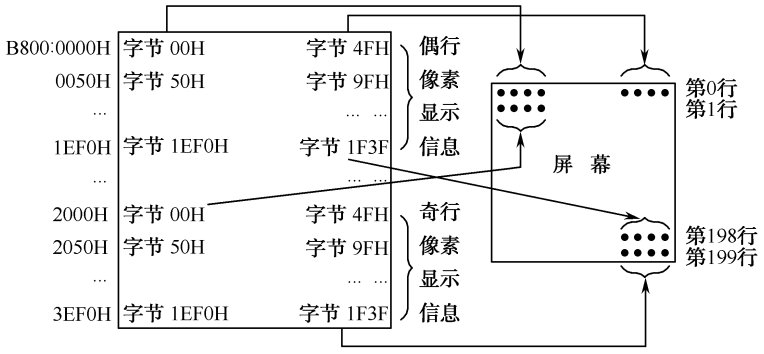


图 9.23　4 色显示缓冲区存储方式示意图

③ 256 色 320×200 图形显示模式。表达 256 种不同颜色需要 8 位二进制，即一个字节。在该模式下，其显示缓冲区的存储方式是非常简单的，即：第一个字节存储第一个像素的颜色，第二个字节存储第二个像素的颜色，以此类推。所以，存储满屏像素所需要的字节数为：320×200×1＝64000。其显示缓冲区的存储方式如图 9.25 所示。

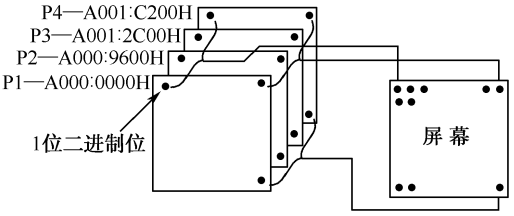


图 9.24　16 色显示缓冲区存储方式示意图

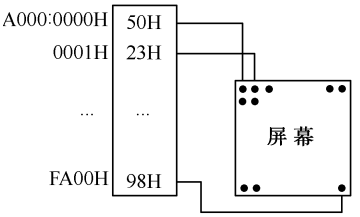


图 9.25　256 色显示缓冲区存储方式示意图

注意：

一般不使用直接操作显示缓冲区的办法来达到改变显示像素的目的，最好是通过 BIOS 内的中断功能来实现相应的功能，这样，所编写的程序能很方便地适应不同的图形显示模式。

2. BIOS 中的屏幕操作功能

BIOS 系统提供了中断 10H 来实现各种屏幕处理功能。在 10H 中断类型中共有 25 个重要的中断服务，如表 9.9 所示。在调用显示器中断服务时，必须把相应的功能号送到 AH 寄存器，然后设置有关的输入参数。表 9.10 列出了常用的几种显示方式。

表 9.9　10H 号中断处理程序的基本功能

AH	功　能	输入参数	返回结果	说　明
00H	设置显示模式	AL=显示模式代号		可设置的显示模式与显示适配卡有关
01H	置光标类型	CH 低 4 位=光标开始线 CL 低 4 位=光标结束线		文本模式下光标大小；当第 4 位为 1 时光标不显示
02H	置光标位置	BH=显示页号 DH=行号 DL=列号		左上角坐标是(0，0)
03H	读光标位置	BH=显示页号	CH=光标开始行 CL=光标结束行 DH=行号 DL=列号	CH 和 CL 含光标类型 左上角坐标是(0，0)
05H	选择当前显示页	AL=新页号		
06H	向上滚屏	AL=上滚行数 BH=填空白行的属性 CH=窗口左上角行号 CL=窗口左上角列号 DH=窗口右下角行号 DL=窗口右下角列号		①当滚动行数为 0 时表示清除整个窗口 ②仅影响当前显示页
07H	向下滚屏	AL=下滚行数 其他参数同 AH=6		同 AH=6
08H	读取光标位置处的字符和属性	BH=显示页号	AH=属性 AL=字符代码	
0AH	将字符写到光标位置处	BH=显示页号 AL=字符代码 BL=属性 CX=字符重复次数		光标不移动
09H	将字符和属性写到光标位置处	BH=显示页号 AL=字符代码 CX=字符重复次数		①光标不移动 ②不带属性
0EH	TTY 方式显示	BH=显示页号 AL=字符代码		光标处显示字符并后移光标；解释回车、换行、遇格和响铃等控制符
0FH	取当前显示模式		(AL)=显示模式号 (AH)=最大列数 (BH)=当前页号	

续表

AH	功　能	输 入 参 数	返 回 结 果	说　明
13H	写字符串（在 AT 及其以上系统上才有此功能）	AL=写模式 BH=显示页号 BL=属性 CX=串中的字符数 DH=写串的起始行号 DL=写串的起始列号 ES：BP=要写串首地址		①解释回车等控制符 ②写模式仅低 2 位： 位 0:0 表示移动光标，1 表示不动光标 位 1:0 表示串中字符代码和属性交替，1 表示串中只含字符代码

表 9.10　常用的显示方式

AH（功能号）	AL（显示模式号）	分　辨　率	显 示 方 式
00H	0	40 列×25 行	黑白文本方式
00H	1	40 列×25 行	彩色文本方式
00H	2	80 列×25 行	黑白文本方式
00H	3	80 列×25 行	彩色文本方式
00H	4	320 列×200 行	彩色图形方式
00H	5	320 列×200 行	黑白图形方式
00H	6	640 列×200 行	黑白图形方式
00H	7	80 列×25 行	黑白文本方式

下面对常见的 INT 10H 的子功能分别介绍。

（1）AH=00H，设置显示模式

输入参数：AL=显示模式号。

返回结果：无返回参数，只是屏幕设置为指定的模式。

例如：将显示器设置为 80×25 的黑白文本方式，16 级灰度。代码如下：

```
MOV     AH, 00H
MOV     AL, 2
INT     10H
```

（2）AH=01H，设置光标类型

输入参数：CH 低四位=光标的起始行；

　　CL 第四位=光标的终止行。

返回结果：无。

例如：把光标扩大到最大。

```
MOV     AH, 01H
MOV     CX, 000CH          ; 光标顶值 0 送 CH，光标底值 0CH 送 CL
INT     10H
```

（3）AH=02H，设置光标位置

输入参数：BH=显示页号（一般为 0）；

　　DH=新光标的行号；

DL=新光标的列号。

返回结果：无。

例如：将光标定位到 0 号页面的 5 行 10 列，代码如下：

```
MOV     AH, 02H
MOV     BH, 0                ; 页号送 BX
MOV     DX, 050AH            ; 行号 5 送 DH，列号 10 送 DL
INT     10H
```

（4）AH=06，屏幕初始化或向上滚动

输入参数：AL=要滚动的行数，AL=0 则全屏滚动（清屏）；

CH=滚动窗口左上角行号，CL=滚动窗口左上角列号；

DH=滚动窗口右下角行号，DL=滚动窗口右下角列号；

BH=滚入行属性。

返回结果：无。

例如：将 25 行 80 列的屏幕上滚 1 行。代码如下：

```
MOV     AX ,0601H            ; 1 送 AL 请求上滚 1 行
MOV     CX, 0                ; 滚动区左上角为 0 行 0 列
MOV     DX, 184FH            ; 滚动区右下角为 24 行 79 列
MOV     BH, 7                ; 属性黑底白字
INT     10H
```

（5）AH=0AH，在光标位置只显示字符，不改变属性

输入参数：BH=显示页；AL=待显示的字符；CX=字符重复次数（只显示一个字符时，CX=1）。

返回结果：无。

例如：在 15 行 20 列位置显示字符 A。代码如下：

```
MOV     AH, 0FH
INT     10H                  ; 取页号：BH←当前页号
MOV     AH, 2
MOV     DX, 1520H
INT     10H
MOV     AH, 0AH
MOV     AL, 'A'
MOV     CX, 1
INT     10H
```

（6）AH=13H，显示字符串

输入参数：ES:BP=字符串首地址；

CX=串长度；

DH，DL=起始行，列号；

BH=显示页号；

AL=0，BL=属性，光标保持在原处，串由字符组成，仅显示字符；

AL=1，BL=属性，光标到串尾，串由字符组成，仅显示字符；

AL=2，光标保持在原位不动，串由字符及属性组成；

AL=3，光标到串尾，串由字符及属性组成。

返回结果：无。

【例 9.8】 编写一个程序，要求功能为：在屏幕中间部位开出一个窗口，随后接受用户按键，并把按键字符显示在窗口的最低行，当窗口底行显示满时，窗口内容就向上滚动一行，用户按 Ctrl+C 组合键时，结束运行。

（1）分析问题

此题主要使用 BIOS 中断 10H 中断 05H 号功能调用置当前显示页面、06H 号功能调用将屏幕初始化或上卷和 02H 功能调用置光标位置。

（2）流程图

流程图如图 9.26 所示。

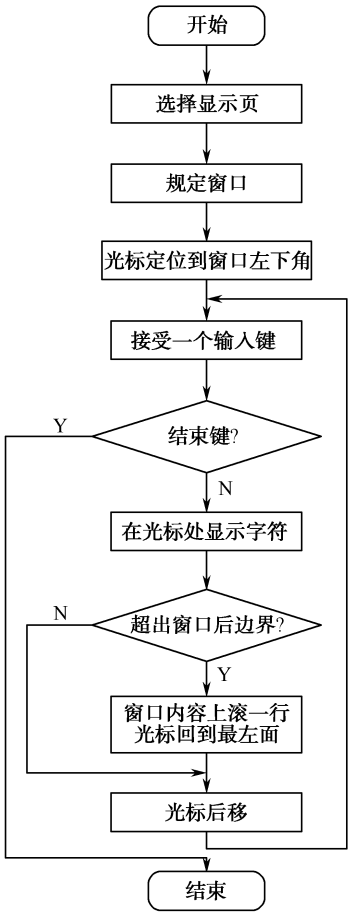


图 9.26 例 9.8 流程图

（3）编写程序

程序如下：

```
WINWIDTH    =  40     ; 窗口宽度
WINTOP      =  8      ; 窗口左上角行号
WINLEFT     =  20     ; 窗口左上角列号
WINBOTTOM   =  17     ; 窗口右下角行号
```

```
 WINRIGHT    =  59     ; 窗口右下角列号
COLOR        =  74H    ; 属性值
PAGEN        =  0      ; 显示页号
CTRLC        =  03H    ; 结束符 ASCII 码
CSEG         SEGMENT
             ASSUME  CS：CSEG
START:       MOV     AL,PAGEN ; 选择显示页
             MOV     AH,05H
             INT     10H
             MOV     CH,WINTOP; 规定窗口
             MOV     CL,WINLEFT
             MOV     DH,WINBOTTOM
             MOV     DL,WINRIGHT
             MOV     BH,COLOR
             MOV     AL,0
             MOV     AH,06H
             INT     10H
             MOV     BH,PAGEN            ; 定位光标到窗口左下角
             MOV     DH,WINBOTTOM
             MOV     DL,WINLEFT
             MOV     AH,02H
             INT     10H
NEXT:        MOV     AH,0                ; 接受一个键
             INT     16H
             CMP     AL,CTRLC            ; 判是否是结束键
             JZ      EXIT                ; 是，转
             MOV     BH,PAGEN            ; 在当前光标位置显示所按键
             MOV     CX,1                ; 但没有移动光标
             MOV     AH,0AH              ; 在光标位置显示字符
             INT     10H
             INC     DL                  ; 光标列数加 1,准备向右移动光标
             CMP     DL,WINRIGHT+1       ; 判是否越出窗口右边界
             JNZ     SETCUR              ; 不，转
             MOV     CH,WINTOP           ; 是，窗口内容上滚一行
             MOV     CL,WINLEFT          ; 空出窗口的最底行
             MOV     DH,WINBOTTOM
             MOV     DL,WINRIGHT
             MOV     BH,COLOR
             MOV     AL,1
             MOV     AH,06H
             INT     10H
             MOV     DL,WINLEFT          ; 光标要回到最左面
SETCUR:      MOV     BH,PAGEN            ; 置光标（光标后移）
             MOV     AH,02H
             INT     10H
```

```
                JMP     NEXT                ；继续
EXIT:           MOV     AX,4C00H
                INT     21H                 ；结束
CSEG            ENDS
                END     START
```

（4）运行程序

程序编译连接后生成可执行文件“EX9_7.exe”文件，运行结果如图 9.27 所示。

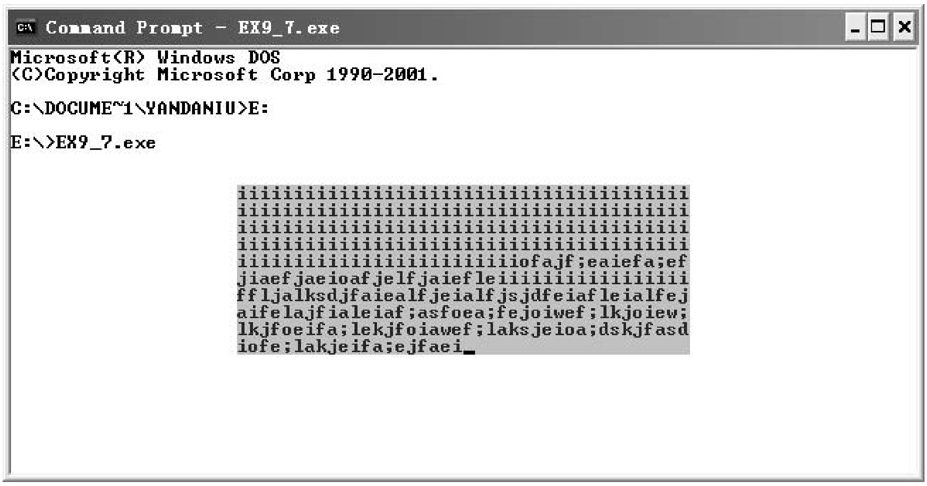


图 9.27　例 9.8 运行结果

9.4.5　显示器 DOS 功能调用

DOS 的 INT 21H 提供的显示功能调用主要有以下 3 个。

1. 02H 号功能调用

功能是显示一个字符，光标跟随字符移动，检测 Ctrl+Break 键。

输入参数：AH=02H，DL 为待显示字符的 ASCII 码。

返回结果：无。

调用格式：

```
MOV     DL, X                   ; X 为要输出显示的 ASCII 字符代码
MOV     AH, 02H
INT     21H
```

2. 06H 号功能调用

功能是显示一个字符，光标跟随字符移动，但不检测 Ctrl+Break 键。

输入参数：AH=06H，DL 为待显示字符的 ASCII 码。

返回结果：无。

3. 09H 号功能调用

功能是显示字符串，光标跟随字符串移动。

输入参数：AH=09H，DS:DX 为待显示字符串的首地址，字符串必须以“$”结尾。

返回结果：无。

【例 9.9】　编写一个程序，用 INT 21H 的 9 号功能显示一个字符串，首先要在数据段中定义一个待显示的字符串，这个字符串必须以“$”结尾。

程序如下：

```
DATA    SEGMENT
STRING  DB      'Welcome',0DH, 0AH, '$'
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX, DATA
        MOV     DS, AX
        LEA     DX, STRING              ; 将 STRING 的偏移地址送入 DX 寄存器中
        MOV     AH,09H                  ; 字符串显示功能，09H 号系统功能调用
        INT     21H                     ; 系统调用
        MOV     AH, 4CH                 ; 返回 DOS
        INT     21H
CODE    ENDS
        END     START
```

说明：

程序中的 0DH 和 0AH 分别表示回车和换行的 ASCII 码，这样就可以自动换行了。

9.4.6　打印机 BIOS 功能调用

BIOS 提供的打印程序作为 17H 号中断处理程序存在。在调用打印程序的某个功能时，应根据要求设置好入口参数，把功能编号置入 AH 寄存器，然后发出中断指令“INT 17H”。除保存出口参数的寄存器外，其他寄存器内容保持不变。通常系统只连接一台打印机，打印机号为 0。INT 17H 提供的功能调用主要有以下 3 个。

1. 00H 号功能调用

功能是将 AL 中的字符打印出来。

入口参数：AH=00H，(DX)为打印机号，(AL)为打印字符。

返回结果：无。

2. 01H 号功能调用

功能是对指定的打印机初始化。

入口参数：AH=01H，(DX)为打印机号。

返回结果：(AH)为状态信息。

3. 02H 号功能调用

功能是读取打印机的状态信息。

入口参数：AH=02H，(DX)为打印机号。

返回结果：(AH)为状态信息。

9.4.7　时钟 BIOS 和 DOS 功能调用

在计算机系统中，有一个可编程定时/计数器 8253 或 8254，它内部有 3 个独立的计数

器，其中计数器 0 作为定时器，为系统时钟提供一个恒定的时间基准，通过对时间基准信号进行计算，就可以获得当前的时间和日期。DOS 和 BIOS 中都提供了用于读取和设置系统时间和日期的功能调用。

需要注意的是，在 BIOS 时间功能调用中，设置的输入参数和返回结果都是以压缩 BCD 码的形式表示的；而在 DOS 时间功能调用中，输入参数和返回结果都是以二进制或者十六进制数表示。

习题

1．简述 I/O 端口的概念。8086/8088 的 I/O 端口地址空间有多大？怎样划分的？

2．说明指令“OUT 20H,AL”和如下程序片段的区别。

```
MOV     DX,20H
OUT     DX,AL
```

3．微机系统可采用哪些方式实现输入/输出？

4．什么是中断？什么是中断源？

5．简述中断的优缺点。

6．中断与子程序调用之间的主要区别是什么？

7．什么是中断的优先级？优先级的处理原则是怎样的？有哪几种解决优先权的办法？

8．什么是中断向量？什么是中断向量表？8088/8086 微机的中断向量表有多大？安排在哪里？

9．简述 BIOS 中断与 DOS 中断及其区别。

10．简述中断的过程。

11．外部中断和内部中断有什么区别？

12．可屏蔽中断和非屏蔽中断有什么区别？

13．对于 256 色，320×200 的显示模式需要多少个字节的内存存放一屏的像素?

14．编写一个清屏程序段。

15．编写将光标置在第 12 行、第 11 列的程序段。

16．编写一个程序段，在屏幕的右下角闪烁显示一段信息。

17．编写一个中断处理程序，中断类型号为 50，完成的功能是将输入的十进制数转换为八进制输出。要求采用不同的方法将中断服务程序的入口地址注册到中断向量表中，然后测试该中断服务。

第 10 章

简单应用程序设计

在 8086/8088 指令集合中，有专门的字符串操作指令，利用它们可有效地进行字符串操作。指令集还有十进制数算术运算调整指令，利用它们可快速实现十进制数算术运算。本章主要介绍串操作指令及串操作的应用、算术运算调整指令及其应用，以及代码转换等。

10.1 汇编语言的字符串

在汇编语言中称存放在内存储器中的成片连续存储的数据为串，如果数据类型是字节，则称做字节字符串；如果数据类型是字，则称做字符串，在不引起混淆的情况下统称为字符串。

例如：

```
S1    DB   'HELLO WORLD!'
S2    DB   '7','8','3','6','4'
S3    DW    5,6,8,7,9,4
S4    DW    80    DUP(?)
S5    DB    90    DUP(0)
```

以上变量定义的都是串。其中 S1、S2、S5 是用 DB 定义的字节串，S3 和 S5 是用 DW 定义的字串。

10.1.1 字符串及其操作命令

在汇编语言中，大量的操作是针对 ACSII 字符串的操作进行的，而对字符串的操作包括复制、检索、插入、删除和退换等，为了方便实现字符串的操作，简化程序设计，8086/8088 提供了专门用于处理字符串的操作指令，称为字符串操作指令，简称为串操作指令。共有五种基本的串操作指令。每种基本的串操作指令包括两条指令，一条适用于以字节为单元的字符串，另一条适用于以字为单元的字符串。串操作指令如下：

- MOVS：传送字节/字/串指令。
- SCAS：搜索字节/字/串指令。

- CMPS：比较字节/字/串指令。
- LODS：取字节/字/串指令。
- STOS：存字节/字/串指令。

串操作指令完成对字符串的各种操作，操作是对一个一个的字符（称为元素）而言的。因此，每一次操作都是针对一个字或字节进行的。由变址寄存器 SI 指向源操作数（串），由变址寄存器 DI 指向目的操作数（串）。规定源串存放在当前数据段中，目的串存放在当前附加段中。在涉及源操作数时，引用数据段寄存器 DS，在涉及目的操作数时，引用附加段寄存器 ES。也就是说，DS:[SI]指向源串，ES:[DI]指向目的串。

串操作指令执行时会自动调整作为指针使用的寄存器 SI 或 DI 的值。如串操作的单元是字节，则调整值为 1；如串操作的单元是字，则调整值为 2。此外，字符串操作的方向（处理字符串中单元的次序）由标志寄存器中的方向标志 DF 控制。当方向标志 DF 复位（为 0）时，按递增方式调整寄存器 SI 或 DI 的值；当方向标志 DF 置位（为 1）时，按递减方式调整寄存器 SI 或 DI 的值。为此，8086 处理器有以下两条设置方向标志的指令：

- CLD（Clear Direction Flag）：置 DF=0。
- STD（Set Direction Flag）：置 DF=1。

为了便于对串中多个字或字节单元进行连续的操作，指令系统还设置有重复前缀指令，重复次数由 CX 中的内容确定。

1. 取串指令（Load String）

取串指令是把有 SI 指定的数据段中某单元[SI]的内容送入 AL 中或 AX 中，并根据方向标志及数据类型修改 SI 的内容。其格式有如下三种。

格式：

```
LODS    SRC
LODSB                ; 字节装入指令，装入字节(Byte)
LODSW                ; 字装入指令，装入字(Word)
```

其中，SRC 表示的是操作数。

如果是字节操作，则先将由 SI 指定的单元内容送入 AL 中，然后 SI 再根据方向标志 DF 复位或置位自加/减 1；如果是字操作，则将[SI]送入 AX 中，然后 SI 再根据方向标志 DF 复位或置位自加/减 2。字符串装入指令的源操作是存储操作数，所以引用数据段寄存器 DS。该指令不影响标志位。

字节装入指令 LODSB 类似于如下的两条指令：

```
MOV     AL, [SI]
INC     SI      或      DEC     SI
```

字装入指令 LODSW 类似于如下的两条指令：

```
MOV     AX, [SI]
ADD     SI, 2       或      SUB     SI, 2
```

LODS 指令将 SI 指向的源串元素根据其地址表达式的属性来决定读取一个字节或字，将其传送到 AL 或 AX，同时根据 DF 和串的类型修改 SI 的内容使之指向串的下一个元素。

例如：

```
……
MSG     DB   'I Love NanJing',0
TAB     DW   345,897,12,56,32
……
MOV     SI, OFFSET MSG
LODS    MSG                          ; LODSB
MOV     SI, OFFSET TAB
LODS    TAB                          ; LODSW
……
```

注意：

操作数 SRC 不影响指针寄存器 SI 的值，所以在使用上述格式的串装入指令时，仍必须先给 SI 赋合适的值。

2. 存串指令（Store String）

存串指令是把 AL 或 AX 的内容（源串元素）存入由 DI 指定的附加段 ES 的某单元（目的串元素）中，并根据 DF 的值及数据类型修改 DI 的内容。其指令格式有如下三种：

格式：

```
STOS    DST          ; 源串元素隐含在 AL 或 AX 中，目的串元素由 ES:DI 指定
STOSB                ; 送字节到目的串
STOSW                ; 送字到目的串
```

其中，DST 为目的操作数。

如果是字节操作，则先将 AL 的内容存入 DI 指定的附加段的某单元[DI]中，然后 DI 再根据方向标志 DF 复位或置位自加/减 1；如果是字操作，则将 AX 的内容存入 DI 指定的附加段的某单元[DI]，然后 DI 再根据方向标志 DF 复位或置位自加/减 2。字符串存储指令的源操作是累加器 AL 或 AX，目的操作是存储操作数，所以引用当前附加段寄存器 ES。该指令不影响标志位。

字节存储指令 STOSB 类似于如下的两条指令：

```
MOV     ES:[DI], AL
INC     Dl     或     DEC     DI
```

字存储指令 STOSW 类似于如下的两条指令：

```
MOV     ES:[DI], AX
ADD     DI, 2     或     SUB     DI, 2
```

在汇编语言中，两条 STOSB 和 STOSW 指令可以使用“STOS DST”统一。汇编程序根据操作数 DST 的类型决定使用字节存储指令还是字存储指令。操作数 DST 不影响指针寄存器 DI 之值。

例如：

```
…
            CLD                  ; 清方向标志，以便按增值方式调整指针
            PUSH    DS           ; 由于在当前数据段中传送数据
            POP     ES           ; 所以使 ES 等于 DS
```

```
        MOV     SI,1000H        ; 置源串指针初值
        MOV     DI,3000H        ; 置目的串指针初值
        MOV     CX,50           ; 置循环次数
NEXT:   LODSB                   ; 取一字节数据
        STOSB                   ; 存一字节数据
        LOOP    NEXT            ; 循环 50 次
…
```

上面程序片段是将当前数据段中偏移 1000H 开始的 50 个字节的数据传送到从偏移 3000H 开始的单元中。

3. 串传送指令（Move String）

串传送指令可以把由 SI 指向的数据段 DS 中的一个字（或字节）送到由 DI 指向的附加段 ES 中的一个字（或字节，目的串）中，同时根据方向标志 DF 及数据格式（字或字节）对 SI 和 DI 进行修改。其指令格式有如下三种。

格式：

```
MOVS    DST,SRC         ; DST 代表目的串元素，SRC 代表源串元素
MOVSB                   ; 字节传送
MOVSW                   ; 字传送
```

其中，DST 代表目的串元素，SRC 代表源串元素，并且数据类型一致。

如果是字节操作，则 SI 与 DI 变化时为加/减 1；如果是字操作，则 SI 与 DI 变化时为加/减 2。根据 DS 和 SI 计算源操作数地址，根据 ES 和 DI 计算目的操作数地址，该指令不影响标志位。

在汇编语言中，两条 MOVSB 和 MOVSW 指令可以用“MOVS DST,SRC”统一。汇编程序根据操作数的类型决定使用字节传送指令还是字存储指令。操作数 DST 或 SRC 可起到方便阅读程序的作用，但不影响寄存器 SI 和 DI 的值，所以在使用上述格式的串传送指令时，仍必须先给 SI 和 DI 赋合适的值。

例如：

```
…
        CLD                     ; 清方向标志
        PUSH    DS              ; 由于在当前数据段中传送数据
        POP     ES              ; 所以使 ES 等于 DS
        MOV     SI,1000H        ; 置源串指针初值
        MOV     DI,3000H        ; 置目的串指针初值
        MOV     CX,50           ; 置循环次数
NEXT:   MOVSB                   ; 每次传送一字节数据
        LOOP    NEXT
…
```

上面程序片段也是将当前数据段中偏移 1000H 开始的 50 个字节的数据传送到从偏移 3000H 开始的单元中。

4. 串比较指令（Compare String）

串比较指令是把由 SI 指向的数据段中的一个字（或字节）与由 DI 指向的附加段中的一个字（或字节）相减，相减结果反映到各有关标志位（AF、CF、OF、PF、SF 和 ZF），但不保存结果，只根据结果置条件码。其指令格式有如下三种。

格式：

```
CMPS     SRC, DST
CMPSB                          ; 字节比较，省略源串和目的串不写
CMPSW                          ; 字比较，省略源串和目的串不写
```

其中，

（1）SRC 是源操作数，表示源串，DST 为目的操作数，表示目的串。

（2）源串元素在由 DS:SI 指定的存储单元，目的串元素在由 ES:DI 指定的存储单元。

（3）源操作数和目的操作数类型一致。

串比较指令每次操作后，自动修改 SI 和 DI。如果是字节操作，则 SI 和 DI 分别自动加/减 l；如果是字操作，则 SI 和 DI 分别自动加/减 2。该指令影响 SF、ZF、AF、PF、CF 和 OF 标志位。

例如：

```
DATA     SEGMENT
SOURCE   DW  87H
TARGET   DW  45H
DATA     ENDS
…
         MOV     SI, OFFSET   SOURCE      ; SI 指向 SOURCE
         MOV     DI, OFFSET   TARGET      ; DI 指向 TARGET
         CMPSW                            ; 比较字
         JA      L1                       ; 如果 SOURCE>TARGET 则跳转到 L1
         JMP     L2                  ; 如果 SOURCE≤TARGET 则跳转 L2
…
```

上例中进行字比较，根据比较结果跳转到 L1 或 L2。

在汇编语言中，两条 CMPSB 和 CMPSW 指令可以用“CMPS SRC,DST”统一。其中源操作数 SRC 或目的操作数 DST 不影响寄存器 SI 和 DI 的值，以及段寄存器 DS 和 ES 的值。

注意：

串比较指令与一般比较指令 CMP 有一个很重要的区别，串比较指令在比较时是源操作数减目的操作数，而一般比较指令是目的操作数减源操作数。

5. 串搜索指令（Scan String）

串搜索指令是把 AL（或 AX）的内容与由 DI 指定的在附加段中的一个字节（或字）相减，相减结果反映到各有关标志位（AF、CF、OF、PF、SF 和 ZF），不保存结果，只根据结果置条件码。其指令格式有如下三种。

格式：

```
SCAS     DST
SCASB              ; 串字节搜索，省略源串和目的串不写
SCASW              ; 串字搜索，省略源串和目的串不写
```

其中，DST 是目的操作数，表示目的串。

串搜索指令每次操作后，自动修改 DI。如果是字节操作，则 DI 自动加/减 1；如果是字操作，则 DI 自动加/减 2。

例如，下面代码是判断字符串 STRING 中是否有字符 D。

```
DATA    SEGMENT
STRING  DB      'ABCDEFGHIGK'
COUNT   EQU     $-STRING
DATA    ENDS
        MOV     AX,DATA
        MOV     ES,AX
        MOV     DI, OFFSET  STRING      ; DI 指向 STRING
        MOV     AL, 'D'                 ; 需要查询的字符 D
        MOV     CX, COUNT               ; 设置查找计数器
        CLD                             ; 方向向前
NEXT:   SCASB
        LOOPNZ  NEXT                    ; 不相等则重复
        JNZ     QUIT                    ; 没找到，转 QUIT
FOUND:  …                               ; 找到
…
QUIT    …                               ; 退出
…
```

在汇编语言中，两条 SCASB 和 SCASW 指令可以用“SCAS DST”统一。其中源由 AL 或 AX 指定，目的串隐含在由 ES:DI 指定的存储单元。汇编程序根据操作数的类型决定使用串字节搜索还是串字搜索。

【例 10.1】 编写程序，若在一片内存储区中有一字符串，起始地址为 STRING。字符串中有字母和数字两种符号，要求把其中的字母和数字符号分开，分别放在同一段的两个空间，存放字母的空间起始地址为 ZIFU；存放数字符号的空间起始地址为 SHUZI。

（1）分析问题

可设 SI 为原字符串指针，设 DI 和 BX 分别为存放数字和字母的目标区指针，使用 LODS 指令把原数据取至 AL 中，然后测试其是字母还是数字符号，若是字母，则用 STOSB 指令送到字母区；若是数字，则可以先把 DI 与 BX 交换，然后再用 STOSB 指令送至数字区，最后再将 DI 与 BX 的内容交换过来。用 CX 来控制循环次数。

（2）流程图

流程图如图 10.1 所示。

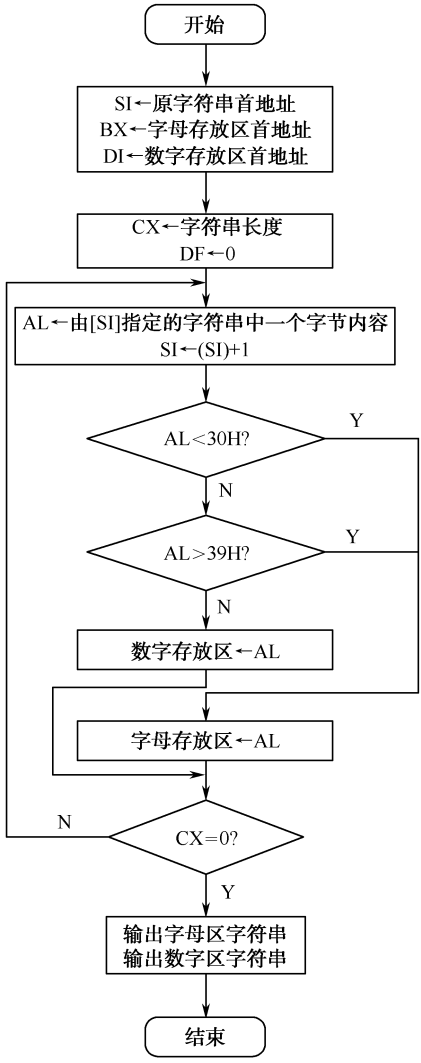


图 10.1　例 10.1 的流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
STRING  DB      'DIDEfe234fef38fj2jf9DE99af3FAE8E83F92fjIjEIFIFQ' ; 源字符串
COUNT   DW      $-STRING                                          ; 字符串长度
ZIFU    DB      50 DUP(?),0DH,0AH,'$'                             ; 字母存放区
SHUZI   DB      50 DUP(?),0DH,0AH,'$'                             ; 数字存放区
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE, DS:DATA, ES:DATA
START:  MOV     AX,DATA
        MOV     DS,AX                       ; DS 指向 DATA
        MOV     ES,AX                       ; ES 指向 DATA
```

```
        MOV     SI,OFFSET STRING            ; DS：SI 指向原字符串
        MOV     BX,OFFSET ZIFU              ; BX 指向字母存放区
        MOV     DI,OFFSET SHUZI             ; DI 指向数字存放区
        MOV     CX, COUNT
        CLD                                 ; 清 DF 为 0（正向串）
LOP:    LODSB                               ; 将原字符串中一个字节取至 AL，且 SI 加 1
        CMP     AL,30H                      ; 判断所取字节为数字符号吗？
        JB      LZIFU                       ; 不为数字符号时转字母处理
        CMP     AL,39H
        JA      LZIFU                       ; 不为数字符号时转字母处理
        STOSB                               ; 是数字符号送数字区，且 DI 加 1
        JMP     AGAIN
LZIFU:  XCHG    BX,DI                       ; 交换 BX 和 DI，使 DI 指向字母字符区
        STOSB                               ; 送字母到字母区，指针增加 1
        XCHG    BX,DI                       ; BX，DI 恢复原值
AGAIN:  LOOP    LOP          ；所有符号都处理了吗，没有则转 LOP；否则，顺序执行
        MOV     DX,OFFSET   ZIFU
        MOV     AH,9                        ; 输出字母区
        INT     21H
        MOV     DX,OFFSET   SHUZI
        INT     21H                         ; 输出数字区
        MOV     AH,4CH
        INT     21H                         ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（4）运行程序

运行的结果如图 10.2 所示。



图 10.2　例 10.1 所运行的结果

10.1.2　重复前缀

由于串操作指令每次只能对字符串中的一个字符进行处理，所以需使用一个循环，以便完成对整个字符串的处理。为了进一步提高效率，8086/8088 还提供了重复指令前缀 REP。重复前缀可加在串操作指令之前，它可以使串操作指令重复地执行，重复的次数由 CX 的内容来确定。

1. 重复前缀 REP

重复前缀 REP 作为一个串操作指令的前缀，它重复其后的串操作指令动作。每一次重

复都先判断 CX 是否为 0，如为 0 就结束重复；否则 CX 的值减 l，重复其后的串操作指令。所以当(CX)=0 时，就不执行其后的串操作指令。其指令格式如下。

格式：

```
REP     strpri
```

其中，strpri 为串操作指令。

重复前缀 REP 主要用在串传送指令 MOVS 和串存储指令 STOS 之前，一般不用在 LODSB 或 LODSW 指令之前。

例如：

```
REP     MOVSB
REP     STOSW
```

2. 重复前缀 REPZ/REPE

重复前缀 REPZ 和 REPE 是同一前缀的两种不同写法，REPZ 指令中 Z 表示结果为 0（ZF=1），REPE 指令中 E 表示相等（ZF=1）。作为一个串操作指令的前缀，它重复其后的串操作指令动作。每重复一次，CX 的值减 1，重复执行，一直到(CX)=0 或串操作指令使零标志 ZF=0 为止。重复结束条件的检查是在重复开始之前进行的。其指令格式如下。

格式：

```
REPZ    strpri
REPE    strpri
```

其中，strpri 为串操作指令。

重复前缀 REPZ/REPE 通常与 CMPS 及 SCAS 共同使用。

例如：

```
REPZ    CMPSB
REPE    CMPSW
REPZ    SCASB
REPE    SCASW
```

其重复的次数由寄存器 CX 和 ZF 共同决定，被解释为没有到字符串尾而且相等重复；否则顺序执行。由于串传送指令 MOVS 和串存储指令 STOS 都不影响标志，所以在这些串操作指令前使用前缀 REP 和前缀 REPZ/REPE 的效果是一样的。

3. 重复前缀 REPNZ/REPNE

重复前缀 REPNZ 和 REPNE 是同一前缀的两种不同写法，REPNZ 指令中 NZ 表示结果不为 0（ZF=0），REPNE 指令中 NE 表示不相等（ZF=0）。作为一个串操作指令的前缀，它重复其后的串操作指令动作。每重复一次，CX 的值减 1，重复执行，一直到(CX)=0 或串操作指令使零标志 ZF=1 为止。重复结束条件的检查是在重复开始之前进行的。其指令格式如下：

格式：

```
REPNZ   strpri
REPNE   strpri
```

其中，strpri 为串操作指令。

重复前缀 REPNZ/REPNE 通常与 CMPS 及 SCAS 共同使用。

例如：

```
REPNZ   CMPSB
REPNE   CMPSW
```

```
REPNZ   SCASB
REPNE   SCASW
```

其重复的次数由寄存器 CX 和 ZF 共同决定，被解释为没有到字符串尾而且不相等重复；否则顺序执行。

注意：

（1）重复前缀指令中的 CX 自动减 1 操作，不影响标志位。

（2）重复的字符串处理操作过程可被中断。CPU 在处理字符串的下一个字符之前识别中断。如果发生中断，那么在中断处理返回以后，重复过程再从中断点继续执行下去。但是如果指令前还有其他前缀（如段超越前缀或锁定前缀），中断返回时其他的前缀就不再有效。这是因为 CPU 在中断时，只能“记住”一个前缀，即字符串操作指令前的重复前缀。如字符串操作指令必须使用一个以上的前缀，则可在此之前禁止中断。

【例 10.2】 编写程序，要求从键盘输入一字符串到以 STRING l 为起始偏移地址的字节缓冲区中。试比较该串与字节字符串 STRING2，如果两串相等，显示出“=”符号表示相等，否则显示“ ^ ”符号表示不等。

（1）分析问题

为简化计算，首先比较这两个字符串的长度是否相等。如不相等。则无继续比较的必要，记(BX)=5EH，比较结束。如果两串长度相等，则可用串比较指令 CMPSB 并带重复前缀 REPZ，将两个字符串的相应字符的 ASCII 码值逐个比较，找出第一个不同字符。若未找到，则串相等，记(BX)=3DH；这里 3DH 和 5EH 分别是“=”和“ ^ ”符号的 ASCII 码。

（2）流程图

流程图如图 10.3 所示。

（3）编写程序

程序如下：

```
DATA    SEGMENT
STRING1 DB      20                      ; 字符串长度
        DB      ?                       ; 记录输入字符后，实际读入的字符数
        DB      20 DUP(0)               ; 存放输入的字符，每个字符占一字节
STRING2 DB      'ASSEMBLY'              ; 定义字符串变量 STRING2
COUNT   EQU     $-STRING2               ; 定义字符串长度变量
STRING3 DB      0DH,0AH,'the compare is','$'     ;显示比较结果
DATA    ENDS
CODE    SEGMENT
        ASSUME  DS:DATA,ES:DATA,CS:CODE
START:  MOV     AX,DATA
        MOV     DS,AX                   ; 数据段初始化
        MOV     ES,AX                   ; 扩展段初始化
        LEA     DX,STRING1              ; 将 STRING1 的起始偏移地址送到寄存器 DX
        MOV     AH,10                   ; 10 号功能调用，从键盘输入一串字符至 STRING1 缓冲区
        INT     21H                     ; 执行 10 号功能调用
        MOV     AL,STRING1+1            ; 实际输入的字符数送入寄存器 AL
        CMP     AL,COUNT                ; 比较两串长度是否相等
        JNE     EXIT                    ; 若不相等则转 EXIT
```

```
        LEA     SI,STRING1+2        ; 将输入字符的偏移地址送入寄存器 SI
        LEA     DI,STRING2          ; 将 STRING2 偏移地址送入寄存器 DI 中为比较作准备
        MOV     CX,COUNT            ; 将字符串长度送入寄存器 CX，充当计数器
        REPZ    CMPSB               ; 逐字节比较两串的对应字符是否相等
        JNE     EXIT                ; 若不相等则转到 EXIT
        MOV     BX,3DH              ; 相等，将=ASCII 码送入 BX
        JMP     RETU            ; 无条件转移到语句 RETU
EXIT:   MOV     BX,5EH              ; 不相等，将 ^ 的 ASCII 码送入 BX
RETU:   LEA     DX,STRING3
        MOV     AH,09H
        INT     21H                 ; 显示比较结果
        MOV     DX,BX
        MOV     AH,02H
        INT     21H
        MOV     AH,4CH              ; 返回 DOS 系统
        INT     21H
CODE    ENDS
        END     START
```

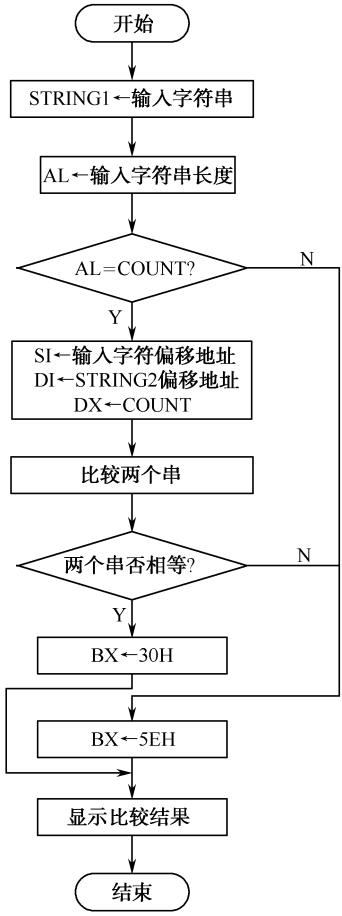


图 10.3　比较字符串是否相等流程图

运行程序

运行的结果如图 10.4 所示。



图 10.4　例 10.2 运算的结果

10.2　十进制数算术运算调整指令及其应用

8086/8088 的十进制数调整指令所认可的十进制数是以 8421BCD 码表示的，8421 码是指 4 个基 2 码的位权从高到底分别为 8、4、2、1，选择的是 0000、0001、0010、…、1001 这 10 种组合，用来表示 0～9 这 10 个数位，如表 10.1 所示。

表 10.1　BCD 码表

十进制数码	8421BCD 码	十进制数码	8421BCD 码
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

这种 8421BCD 码与十进制数关系直观。本书后面提到的 BCD 码都是指 8421BCD 码。BCD 码与十进制数转换也比较简单。例如：求十进制数 57.3 的 BCD 码，其转换过程如下：

$$\begin{array}{cccc} 5 & 7 & . & 3 \\ 0101 & 0111 & . & 0011 \end{array}$$

所以，$(57.3)_{10}=(01010111.0011)_{BCD}$

再如，求 BCD 码 10000011.0111 所对应的十进制数，其转换过程如下：

$$\begin{array}{cccc} 1000 & 0011 & . & 0111 \\ 8 & 3 & . & 7 \end{array}$$

所以，$(10000011.0111)_{BCD}=(83.7)_{10}$

十进制数是逢十进一的，用 BCD 码表示时，是把每 1 位十进制数字用 4 位二进制数表示，是逢十六进一。4 位二进制码有 16 种不同的状态，而 1 位十进制数只有 10 种不同的数字；当 BCD 数按二进制运算时，不可避免会出现 6 种不用的状态，因此采用 BCD 码

后，在两数相加的和小于或等于 9 时，十进制运算的结果是正确的；而当相加的和大于 9 时，结果不正确，必须加 6 修正后才能得出正确的结果。

BCD 码的运算规则是：两个 BCD 码相加，如果和等于或小于 1001B，即 9H，则不需要修正；如果相加之和在 1010B 到 1111B 即 0AH～0FH 之间，则需要 6H 加以修正；如果相加时本位产生了进位，则需加 6H 加以修正。

例如：

利用 BCD 码计算 4+5 的运算过程如下：

```
  0100 4
+ 0101 5
-------
  1001 9
```

利用 BCD 码计算 5+7 的运算过程如下：

```
  0101 5
+ 0111 7
-------
  1100  结果大于 9
+ 0110  加 6 修正
-------
 10001 2
  ↑
进位
```

利用 BCD 码计算 8+9 的运算过程如下：

```
  1000 8
+ 1001 9
--------
 11100  结果大于 9
+ 0110  加 6 修正
--------
 10111 7
```

本节介绍的加法运算的十进制调整指令的基本思想，就是在必要时让运算的结果跳过 6 种不用的状态，其方法是在原结果上加 6，即逢十，就当成十六。这便是用 BCD 数进行十进制运算的正确结果。减法调整的基本思想与加法一样，只是在调整时应减去 6。

BCD 码分为非压缩（未组合）和压缩（组合）两种。压缩 BCD 码是指一个字节存放两位 BCD 码，例如，将十进制数 86 用压缩 BCD 码表示，则为：

```
1 0 0 0  0 1 1 0
---------  -----
    8        6
```

非压缩 BCD 码是指一个字节含一位 BCD 码，字节的高 4 位无意义，通常使用“0000”和“1111”作为标志。例如，将十进制数 86 用非压缩 BCD 码表示，则为：

```
0 0 0 0 1 0 0 0  0 0 0 0 0 1 1 0
-----------------  -------------
        8                6
```

比如，十进制数 1234 的压缩 BCD 码就是 1234H。非压缩 BCD 码是指用 8 位二进制数表示 1 位十进制数，其中低 4 位为 BCD 码，高 4 位随意，但通常取 0。所以十进制数 1234 的非压缩 BCD 码，可表示为 01020304H。两种情况下，BCD 数调整的具体方法有所不同。

另外数字的 ASCII 码是一种非压缩的 BCD 码，因为数字的 ASCII 码的低 4 位是对应的 8421BCD 码。

注意：

压缩 BCD 码只有加减运算，没有乘除运算。

10.2.1 压缩 BCD 码的算术运算调整指令

1. 压缩 BCD 码加法的十进制调整指令 DAA

压缩 BCD 码加法的十进制调整指令 DAA 的功能是，接在指令 ADD（或 ADC）后，对两个“用压缩 BCD 码表示的”2 位十进制数（其中一个一定在 AL 中）相加的结果（放在 AL 中）进行校正，产生一个正确的“用压缩 BCD 码表示的”两位十进制数的和。其格式如下。

格式：

```
DAA
```

指令 DAA 会影响 AF、CF、PF、SF、ZF 标志位，OF 标志位不影响。调整方法为：

（1）如 AL 中的低 4 位在 0～9 之间，或标志位 AF =0，不调整。

（2）如 AL 中的低 4 位在 A～F 之间，或 AF 标志位为 1，则(AL)←(AL)+6，并且 AF 置 1。

（3）如 AL 中的高 4 位在 A～F 之间，或 CF 标志位为 1，则(AL)←(AL)+60H，并且 CF 置 1。

例如：

```
MOV     AL,34H
ADD     AL,47H
DAA
```

上例中第一条指令执行后，AL 为 34H；因为 ADD 是二进制数相加，所以第二条指令执行后 AL 为 7BH，AF 标志位为 0，CF 标志位为 0；第三条指令进行压缩 BCD 码加法的十进制调整，执行后 AL 为 81H，AF 标志位为 1，CF 标志位为 0。

2. 压缩 BCD 码减法的十进制调整指令 DAS

压缩 BCD 码减法的十进制调整指令 DAS 功能是接在指令 SUB（或 SBB）后，对两个“用压缩 BCD 码表示的”2 位十进制数（其中一个一定在 AL 中）相减的结果（放在 AL 中）进行校正，产生一个正确的“用压缩 BCD 码表示的”2 位十进制数的差。其格式如下。

格式：

```
DAS
```

指令 DAS 会影响 AF、CF、PF、SF、ZF 标志位，OF 标志位无定义。所调整方法为：

（1）如 AL 中的低 4 位在 0～9 之间，或标志位 AF =0，不调整。

（2）如 AL 中的低 4 位在 A～F 之间，或者 AF 标志位为 1，则(AL)←(AL)−6，并且 AF 置 1。

（3）如 AL 中的高 4 位在 A～F 之间，或者 CF 标志位为 1，则(AL)←(AL)−60H，并且 CF 置 1。

例如：

```
MOV     AL,45H
SUB     AL,27H
DAS
```

上例中第一条指令执行后，AL 为 45H；第二条指令执行后，因为是 SUB 是二进制数相减，AL 为 1EH，AF 标志位为 1，CF 标志位为 0；第三条指令进行压缩 BCD 码减法的十进制调整，执行后 AL 为 18H，AF 标志位为 1，CF 标志位为 0。

【例 10.3】　编写程序，实现 12345678+23456789，并显示计算结果。

（1）分析问题

将这 2 位十进制数以压缩 BCD 码的形式存储，并且低位放在低地址字节里，一个字节存放两位 BCD 码，所以 8 位十进制需要 4 个字节才能完全表示。将第一个加数存放在以变量名 DATA1 为起始地址的存储单元中，将第二个加数存放在变量名 DATA2 为起始地址的存储单元中。计算过程是从低位字节开始对应字节相加，先用一条 ADC 指令，再用 DAA 指令进行调整。由于用的是 ADC 指令，故第一次相加前必须将 CF 清零才能保证第一个字节相加的和是正确的。将相加的结果存放在 DATA3 为起始地址的存储单元中，因为结果可能比原来的加数多 1 位，故多分配一个字节，初值为 0。

以压缩 BCD 码存储的计算结果低位存放在低地址，因此输出时需要从高地址（高位）开始。因为压缩 BCD 码是一个字节存放 2 位十进制数，所以将一个字节中的 BCD 码显示出时需要将两位 BCD 码分离开来。可以使用 SHR 指令将其右移 4 位，用 OR 指令拼上 30H，使之转化为 ASCII 码显示出来，然后使用 AND 指令将这个字节的低 4 位截取出来并拼上 30H 显示出来。

（2）流程图

流程图如图 10.5 所示。

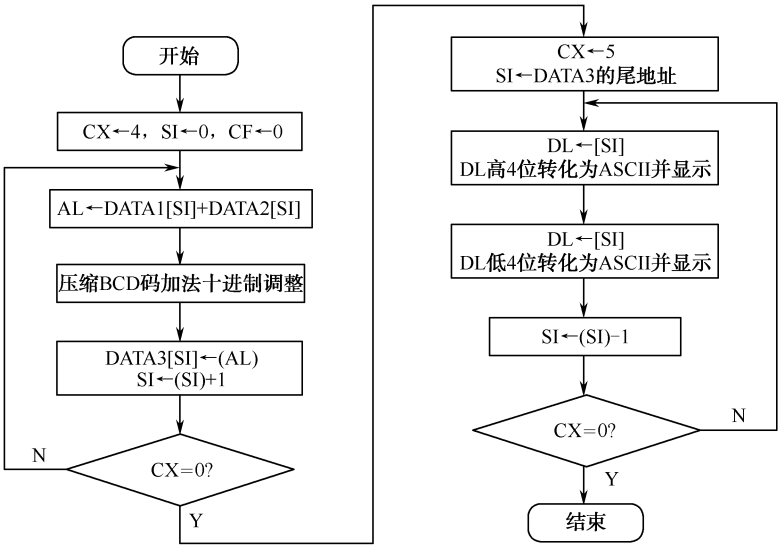


图 10.5　2 位十进制数相加并显示流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
DATA1   DB   78H,56H,34H,12H          ; 加数 1
DATA2   DB   89H,67H,45H,23H          ; 加数 2
DATA3   DB   5 DUP(0)                 ; 存放计算结果
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     CX,4
        MOV     SI,0
        CLC
LOP1:   MOV     AL,DATA2[SI]          ; 开始准备两数相加
        ADC     AL,DATA1[SI]          ; 执行相加
        DAA                           ; 十进制调整
        MOV     DATA3[SI],AL          ; 将计算结果存放在 DATA3 开始的存储区域
        INC     SI
        LOOP    LOP1
        ADC     BYTE PTR[SI],0        ; 保存最高位的进位
        MOV     CX,5                  ; 显示
        MOV     SI,OFFSET DATA3+4
LOP2:   MOV     DL,[SI]
        MOV     BX,CX
        MOV     CL,4
        SHR     DL,CL
        OR      DL,30H
        MOV     AH,02H
        INT     21H
        MOV     DL,[SI]
        AND     DL,0FH
        OR      DL,30H
        MOV     AH,02H
        INT     21H
        DEC     SI
        MOV     CX,BX
        LOOP    LOP2
        MOV     AH,4CH
        INT     21H
CODE    ENDS
        END     START
```

（4）运行程序

程序运行的结果如图 10.6 所示。

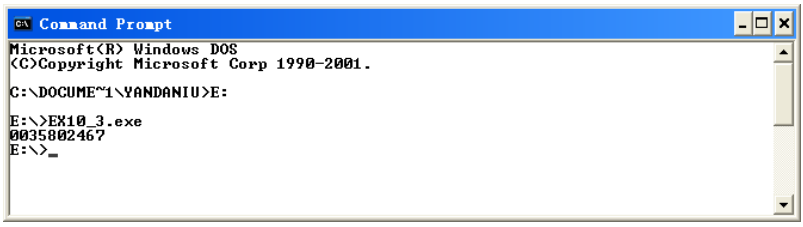


图 10.6　2 位十进制数相加的计算结果

10.2.2　非压缩 BCD 码的算术运算调整指令

1. 非压缩 BCD 码加法的十进制调整指令 AAA

非压缩 BCD 码加法的十进制调整指令 AAA 的功能是，接在指令 ADD（或 ADC）后，对两个“用非压缩 BCD 码表示的”1 位十进制数（其中一个一定在 AL 中）相加的结果（放在 AL 中）进行校正，产生一个正确的“用非压缩 BCD 码表示的”十进制数的和。其格式如下。

格式：

```
AAA
```

指令 AAA 影响 AF 和 CF 标志位，其余标志位均无定义。调整方法为：

（1）如 AL 中的低 4 位在 0～9 之间，且 AF 为 0，则转第（3）步。

（2）如 AL 中的低 4 位在 A～F 之间，或 AF 为 1，则(AL)←(AL)+6，(AH)←(AH)+1，且 AF 标志位置 1。

（3）清除 AL 的高 4 位。

（4）AF 标志位的值送到 CF 标志位。

例如：

```
MOV     AX,7
ADD     AL,6
AAA
```

上例中第一条指令执行后 AX 为 07H；第二条指令执行后 AL 为 0DH，AH 为 00H，AF 和 CF 标志位都为 0；第三条指令进行非压缩 BCD 码加法的十进制调整，执行后 AL 为 03H，AH 为 01H，AF 和 CF 标志位都为 1。

2. 非压缩 BCD 码减法的十进制调整指令 AAS

非压缩 BCD 码减法的十进制调整指令 AAS 的功能是，接在指令 SUB（或 SBB）后，对两个“用非压缩 BCD 码表示的”1 位十进制数（其中一个一定在 AL 中）相减的结果（放在 AL 中）进行校正，产生一个正确的“用非压缩 BCD 码表示的”十进制数的差。其格式如下。

格式：

```
AAS
```

指令 AAS 影响 AF 和 CF 标志位，其余标志位均无定义。调整方法为：

（1）如 AL 中的低 4 位在 0～9 之间，且 AF 标志位为 0，则转第（3）步。

（2）如 AL 中的低 4 位在 A～F 之间，或 AF 标志位为 1，则(AL)←(AL)−6，(AH)←(AH)−1，且 AF 标志位置 1。

（3）清除 AL 的高 4 位。

（4）AF 标志位的值送 CF 标志位。

例如：

```
MOV     AL,00H
SUB     AL,09H
AAS
```

上例中第一条指令执行后，AL 为 34H；第二条指令执行后，AL 为 2BH，AF 标志位为 1，CF 标志位为 0；第三条指令进行非压缩 BCD 码减法的十进制调整，执行后 AL 为 05H，AF 和 CF 标志位都为 1。

3. 非压缩 BCD 码乘法的十进制调整指令 AAM

乘法运算的十进制调整，是对两个一位非压缩 BCD 码而言的；否则，则无法调整。非压缩 BCD 码乘法的十进制调整指令 AAM 接在指令 MUL 后（BCD 码总是表示无符号数，所以相乘时总是用指令 MUL，不用指令 IMUL），乘法十进制调整是将相乘的结果（在 AL 中）除以 0AH（即 10），所得的商放在寄存器 AH 中，余数放在寄存器 AL 中。其格式如下。

格式：

```
AAM
```

指令 AAM 会影响 SF、ZF 和 PF，AF、CF 和 OF 标志位无定义。

例如：

```
MOV     AL,03H                  ; AL=03H,表示非压缩 BCD 码 3
MOV     BL,04H                  ; BL=04H,表示非压缩 BCD 码 4
MUL     BL
AAM
```

上例中前两条指令执行后，AL 和 BL 分别为 03H 和 04H；第三条指令执行后，AL 为 0CH，AH 为 00H；第四条指令进行非压缩 BCD 码乘法的十进制调整，执行后 AL 为 02H，AH 为 01H。

4. 非压缩 BCD 码除法的十进制调整指令 AAD

除法运算的十进制调整，也是对非压缩 BCD 码而言的；否则，则无法调整。非压缩 BCD 码除法的十进制调整指令 AAD 功能是，把存放在寄存器 AH（高位十进制数）及存放在寄存器 AL 中的两位非压缩 BCD 码被除数调整为 1 个二进制数，存放在寄存器 AL 中。其格式如下。

格式：

```
AAD
```

除法运算的十进制调整有所不同，它是在除法运算前进行的，调整的方法为：

```
(AL)←(AH)×0AH+(AL)
(AH)←0
```

这里 AH 中的内容是高位相除时的余数。指令 AAD 影响标志位 SF、ZF 和 PF，对其他标志无定义。

例如：

```
MOV     AH,04H
MOV     AL,03H
MOV     BL,08H
AAD
DIV     BL
```

上例中前 3 条指令执行后，AH、AL 和 BL 分别为 04H、03H 和 08H；第四条指令进

行非压缩 BCD 码除法的十进制调整，执行后 AL 为 2BH，AH 为 00H，第五条指令执行后，AL 为 05H，AH 为 03H。

【例 10.4】　编写程序，实现 12345×9，并显示计算结果。

（1）分析问题

此题是多位 BCD 码与一位 BCD 码进行相乘。12345×9 的通常的算法为：5×9=45，写下 5 同时将 4 进位；4×9=36，加进位 4 得 40，写下 0 同时将 4 进位；3×9=27，加进位 4 得 31，写下 1 同时将 3 进位；2×9=18，加进位 3 得 21，写下 1 同时将 2 进位；1×9=9，加进位 2 得 11，写下 1 同时将 1 进位；所以结果为 111105。写成竖式为：

12345←被乘数
×　　9←乘数
+11234←进位
111105←结果

显示计算结果数据时，应将数字 0～9 转换成 ASCII 码，即使用 OR 指令将其高 4 位拼上 3 即可。

（2）流程图

流程图如图 10.7 所示。

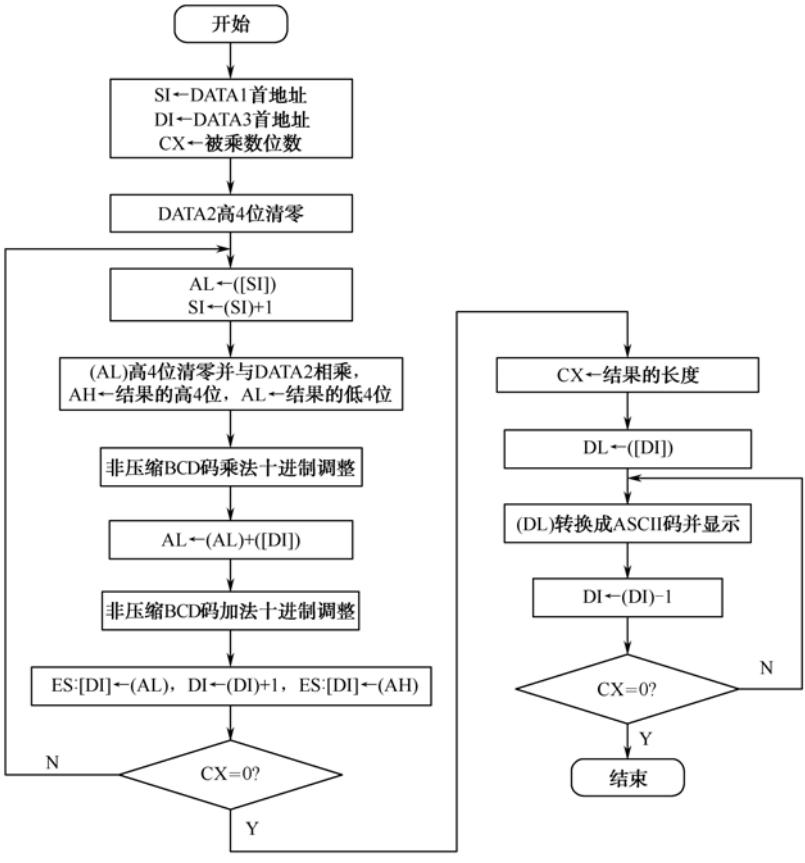


图 10.7　2 位十进制数相乘并显示流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
DATA1   DB   '5','4','3','2','1'          ; 被乘数 1
DATA2   DB   '9'                          ; 乘数 2
DATA3   DB   6 DUP(?)                     ; 存放计算结果
COUNT   EQU $- DATA3                      ; 计算结果的长度
DATA    ENDS
CODE    SEGMENT
        ASSUME   CS:CODE,DS:DATA,ES:DATA
START:  MOV      AX,DATA
        MOV      DS,AX
        MOV      SI,OFFSET DATA1   ; DATA1 首地址→SI
        MOV      DI,OFFSET DATA3   ; DATA2 首地址→DI
        MOV      CX,5              ; 设置循环次数为 5
        AND      DATA2,0FH         ; 将 DATA2 的高 4 位清零
        MOV      BYTE PTR[DI],0
CHENG:  MOV      AL,[SI]           ; 取 DATA1 的一个字节送入 AL
        INC      SI                ; SI 加 1
        AND      AL,0FH            ; 将 AL 的高 4 位清零
        MUL      DATA2             ; AL 与 DATA2 相乘
        AAM                        ; 非压缩 BCD 码乘法十进制调整
        ADD      AL,[DI]           ; 加进位
        AAA                        ; 非压缩 BCD 码加法十进制调整
        MOV      [DI],AL           ; 存入（AL）
        INC      DI                ; DI 加 1
        MOV      [DI],AH           ; 存入（AH）
        LOOP     CHENG
        MOV      CX,COUNT          ; 设置循环次数
SHOW:   MOV      DL,[DI]           ; 取结果的一个字节送入 DL
        OR       DL,30H            ; 转换成 ASCII 码
        MOV      AH,02H            ; 2 号功能调用
        INT      21H               ; 输出结果
        DEC      DI
        LOOP     SHOW
        MOV      AH,4CH
        INT      21H
CODE    ENDS
        END      START
```

（4）运行程序

程序运行的结果如图 10.8 所示。

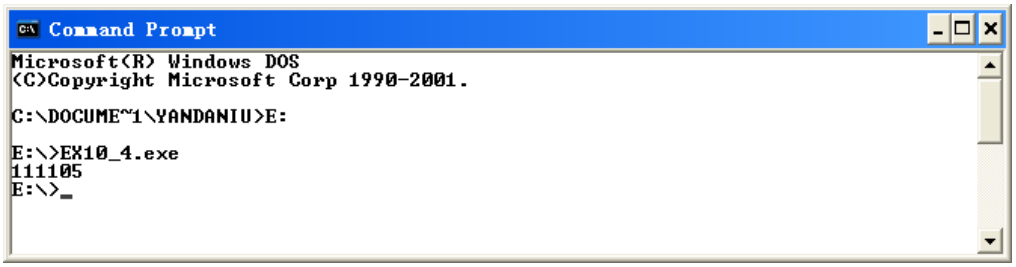


图 10.8　2 位进制数相乘的计算结果

说明：

上例中只介绍了多位数被乘数和一位乘数相乘，但很容易将其推广到多位数和多位数相乘，请读者思考。

10.3　代码转换

代码转换是输入/输出过程中经常遇到的问题，因为在计算机内部只能进行二进制数的运算。输入时，计算机系统要把用户从输入设备提供的数和字符的 ASCII 码形式表达的十进制和其他进制转换成二进制，并存储在内存中；输出时，要把计算机内部的二进制数转换成输出需要的 ASCII 码或者十进制数，然后再输出。

在高级语言编程环境中，程序员能用各种输入/输出语句，按一定的格式进行交互式操作，很少或根本不关心输入/输出是如何实现的。有的程序员甚至认为其输入的就是十进制数值，输出数据也就是把内存中存储的数据直接输出出来。其实，输入/输出过程并不是如此简单，计算机系统要进行复杂而又细致的代码转换和格式化等工作。

10.3.1　其他进制数到二进制数的转换

在进行汇编语言程序设计的时候，经常遇到 ASCII 码表示的十进制数、压缩与非压缩 BCD 表示的十进制数等以特殊格式存储的数向二进制数转换的问题。它们实际是各类存储形式数据向各自实际数值的变换，只有得到实际数值才可以进行加、减及其他形式的运算。

1. 十进制数到二进制数的转换

ASCII 码十进制数是用特殊规定的二进制数表示与存储十进制数的一种方法，例如，用 33H、37H、38H 分别表示十进制数 3、7、8，将它们邻接放到一起，就形象地代表 378。但是，如果要进行 378 与其他数的运算，直接使用 ASCII 码十进制数 33H、34H、38H 是无法进行的，必须变成二进制码才能进行运算。实现该变换的一般方法是将每一位的数字乘以所在位的权值后相加，例如，32H 代表 2，放在百位数位上，就认为它代表 3×100=300（权值为 100），7 代表 70（权值为 10），8 就代表 8（权值为 1）；将 300 +70 +5 就得到 33H、37H、38H 代表的实际数，这是 1 个二进制形式存放的数。由此可以总结出 ASCII 码十进制数转换成二进制数的方法：将每一个 ASCII 码代表的十进制数字乘以该数字所处数位的权后相加。

【例 10.5】编写程序，使其功能为：将(SI)指针所指向的缓冲区 BUF 中保存的（COUNT+1）位 ASCII 码十进制数（假设十进制数的范围为 0～65535）转换成对应的二进制数，并以二进制形式显示出来。

(1) 分析问题

根据类似于 378=(((3×l0) +7)×l0) +8 这样的算法，可以得到主要程序流程如下。

① 将寄存器 SI 指向 BUF，DX 存放中间结果，起初值为 0，BX 存放权值乘数 10，AX 做乘法要求的默认寄存器。AX 初值设为 0。

② 取出 SI 所指的 ASCII 码字符，从其二进制存储数值中减去 30H，得到所代表的十进制数字，即真实值。

③ 将该数字加到 AX 中后乘以权值 10。

④ 修改指针指向下一个数对应的 ASCII 码的存放单元并转到步骤②。

需要注意的是个位不与权值 10 相乘，所以做独立处理。

（2）流程图

流程图如图 10.9 所示。

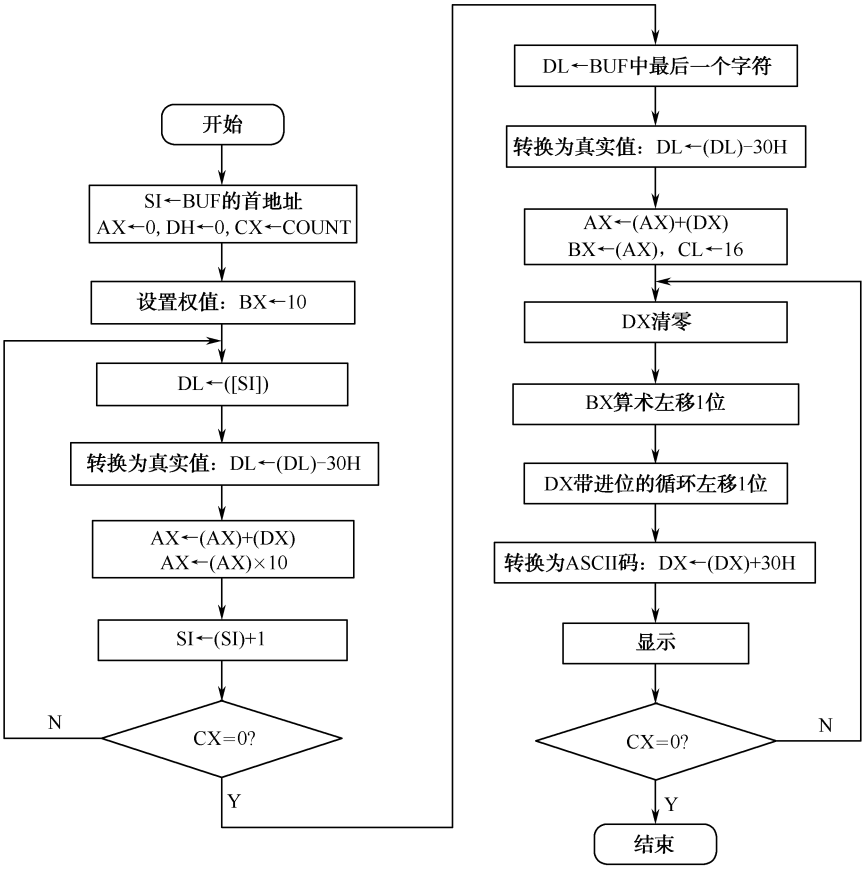


图 10.9　十进制转换二进制并显示流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
BUF     DB      33H,37H,38H
COUNT   EQU     $-BUF-1             ; 数据长度减去 1
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     SI,OFFSET BUF       ; 将 BUF 的首地址送入 SI
        MOV     AX,0
        MOV     DH,0                ; 清 DX 高位
        MOV     CX,COUNT            ; 设置循环次数为 COUNT
        MOV     BX,10               ; 权数 10 存到 BX 中
NEXT:   MOV     DL,[SI]             ; 取出字符
        SUB     DL,30H              ; 将 ASCII 码表示的数字转换为对应的真实值
        ADD     AX,DX               ; 将 AX 加上 DX
        MUL     BX                  ; AX 乘以 10 结果存放在 AX 中
        INC     SI                  ; 使指针 SI 指向下一个字符
        LOOP    NEXT
        MOV     DL,[SI]             ; 取出最后一位字符
        SUB     DL,30H              ; 将 ASCII 码表示的数字转换为对应的真实值
        ADD     AX,DX               ; 将 AX 加上 DX
        MOV     BX,AX               ; 将待显示的结果送入（BX）中
        MOV     CL,16               ; 置(CL)=16,转换后的结果是 16 位的二进制数
SHOW:   MOV     DX,0                ; 逐位显示二进制的计算结果，将 DX 清 0
        SAL     BX,1                ; 算术左移 1 位
        RCL     DX,1                ; 带进位的循环左移 1 位
        ADD     DX,30H              ; 转换为对应的 ASCII 码
        MOV     AH,2                ; 显示
        INT     21H
        LOOP    SHOW
        MOV     AH,4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（4）运行程序

运行结果如图 10.10 所示，378 转换为二进制数为 101111010。



图 10.10　378 转换二进制数结果

2. 十六进制数到二进制数的转换

十六进制数到二进制数的转换和十进制数到二进制数的转换比较相似，只是权值为16，另外，十六进制数的数字字符为 0～9 和 a～f（或 A～F），采用键盘逐个输入后以 ASCII 码 30H～39H 和 61H～66H（或 41H～46H）存入（AL）。因此若是数字键 0～9，则其 ASCII 码减去 30H 就可以得到对应的数值，即是该位十六进制数对应的 4 位二进制值；若是字符键 A～F 或 a～f，其 ASCII 码则要分别减去 37H 和 57H，才能得到输入的十六进制字符对应的数值。表 10.2 列出了十六进制数的 ASCII 码与二进制数之间的转换规律。

表 10.2　十六进制数的 ASCII 码与二进制数转换对照表

范围	十六进制字符	ASCII 码	二进制数	A～F 之间 ASCII 码减 7 后二进制数	a～f 之间 ASCII 码减 7 后二进制数
0～9	0	30H	0000		
	1	31H	0001		
	2	32H	0010		
	3	33H	0011		
	4	34H	0100		
	5	35H	0101		
	6	36H	0110		
	7	37H	0111		
	8	38H	1000		
	9	39H	1001		
A～F	A	41H	1010	0011 1010	
	B	42H	1011	0011 1011	
	C	43H	1100	0011 1100	
	D	44H	1101	0011 1101	
	E	45H	1110	0011 1110	
	F	46H	1111	0011 1111	
a～f	a	61H	1010		0101 1010
	b	62H	1011		0101 1011
	c	63H	1100		0101 1100
	d	64H	1101		0101 1101
	e	65H	1110		0101 1110
	f	66H	1111		0101 1111

【例 10.6】 编写程序，使其功能为：从键盘输入 4 位十六进制数，将其转换为二进制数并显示出来。

（1）分析问题

从键盘上输入的都是字符，即 ASCII 码值，需要将其转换为十六进制数。由前面介绍可知，只需要将 0～9 的 ASCII 码的高 4 位中的 3 去掉，就成为与其对应的十六进制数值；

而 A～F 的 ASCII 码减去 37H 即成为与其对应的十六进制数值；a～f 的 ASCII 码减去 57H 即成为与其对应的十六进制数值。

这里需要注意的是，数据是通过键盘一个个输入的，需要调用 1 号功能中断，要占用 AX 寄存器，这就要求处理好 AX 角色的变换。

（2）流程图

流程图如图 10.11 所示。

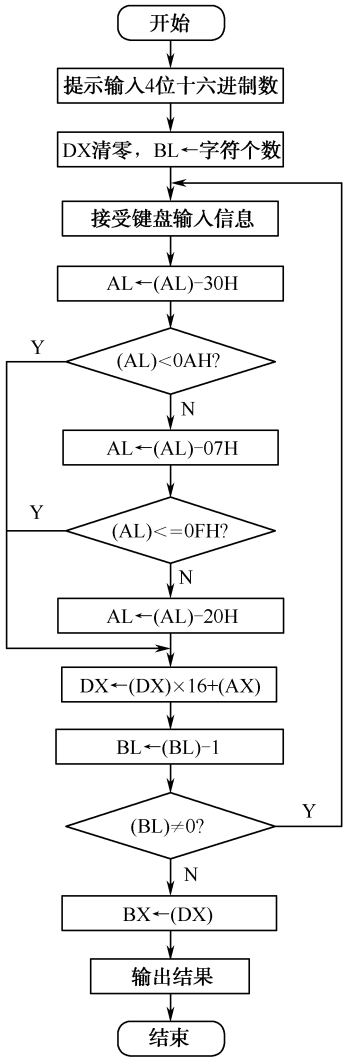


图 10.11　十六进制数转换二进制数并显示流程图

（3）编写程序

程序如下：

```
DATA      SEGMENT
PROMPT  DB   'PLEASE INPUT 4 HEXADECIMAL NUMBER:$'      ; 提示输入信息
RESULT  DB   0AH,0DH,'THE RESULT IS:$'                  ; 提示计算结果信息
DATA      ENDS
```

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA
START:  MOV     AX,DATA
        MOV     DS,AX
        LEA     DX,PROMPT           ; 提示输入 4 位十六进制数
        MOV     AH,09H
        INT     21H
        MOV     DX,0                ; DX 清零
        MOV     BL,4                ; 输入字符个数
INPUT:  MOV     AH,01H              ; 1 号功能调用，接受键盘输入信息
        INT     21H
        SUB     AL,30H              ; (AL)减去 30H
        CMP     AL,0AH              ; (AL)与 0AH 比较
        JL      CHANGE              ; 输入字符在 0～9 范围中，则转到 CHANGE
        SUB     AL,07H
        CMP     AL,0FH
        JLE     CHANGE              ; 输入在 A～F 范围中
        SUB     AL,20H              ; 输入在 a～f 范围中
CHANGE: MOV     CL,4
        SHL     DX, CL              ; 乘 16
        MOV     AH,0
        ADD     DX,AX               ; (DX)←(DX)×16+(AX)
        DEC     BL
        JNZ     INPUT               ; 转输入下一字符
        MOV     BX,DX               ; 将待显示的结果送入（BX）中
        LEA     DX,RESULT           ; 提示输出结果
        MOV     AH,09H
        INT     21H
        MOV     CL,16               ; 置(CL)=16,转换后的结果是 16 位的二进制数
SHOW:   MOV     DX,0                ; 逐位显示二进制的计算结果，将 DX 清 0
        SAL     BX,1                ; 算术左移 1 位
        RCL     DX,1                ; 带进位的循环左移 1 位
        ADD     DX,30H              ; 转换为对应的 ASCII 码
        MOV     AH,2                ; 显示
        INT     21H
        LOOP    SHOW
        MOV     AH,4CH
        INT     21H                 ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（4）运行程序

运行结果如图 10.12 所示，1AC2 转换为二进制数为 1101011000010。

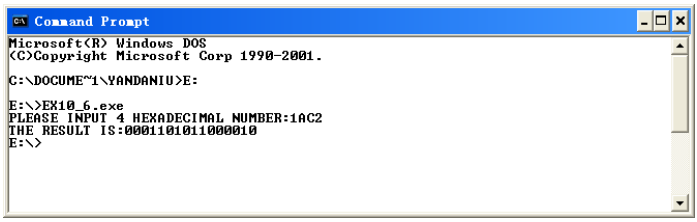


图 10.12　1AC2 转换二进制数结果

3. 压缩 BCD 码到二进制数的转换

【例 10.7】　使编写程序，使其功能为：将 4 位压缩 BCD 码转换为二进制数，并且将结果显示出来。假设压缩 BCD 码为 1001 0010 0110 0100。

（1）分析问题

本例要求首先分离出 SI 中 4 个 BCD 码代表的数，再转换成二进制码。分离 4 个 BCD 码的方法是利用 RCL 循环移位，每次移 4 位到低 4 位，保存移位后的值后，屏蔽掉高 12 位，就得到该 BCD 码代表的数。

（2）流程图

流程图如图 10.13 所示。

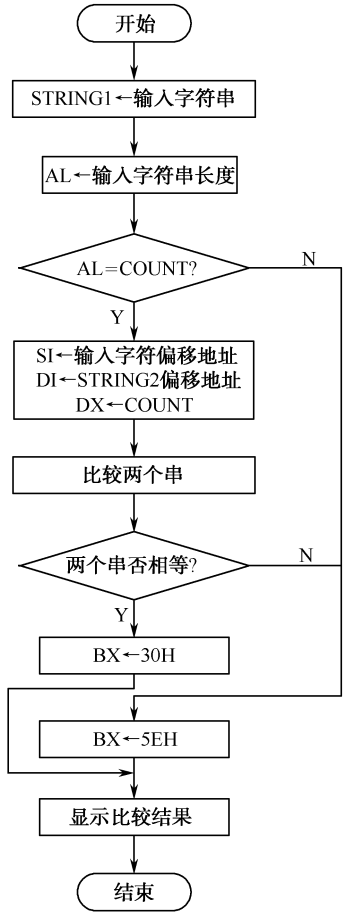


图 10.13　压缩 BCD 码转换二进制数并显示流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
DATA1   EQU     1001001001100100B       ; 压缩 BCD 码二进制
DATA    ENDS
STACK1  SEGMENT     PARA    STACK
        DW      20H DUP(0)
STACK1  ENDS
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     SI,DATA1        ; 将 BCD 码送入 SI
        MOV     CX,4            ; BCD 码共四位
        MOV     DI,10           ; 以 10 为权，作乘数
        MOV     AX,0            ; AX 存放中间结果
NEXT:   MUL     DI              ; (AX)乘 10，结果放在 AX 中
        MOV     BX,SI           ; BCD 码数据放到 BX 中进行移位
        PUSH    CX              ; 循环与移位都用 CX，将循环的 CX 数保存起来
        MOV     CL,4            ; 每个 BCD 码对应 4 位二进制数
        ROL     BX,CL           ; 循环移位，将最高位 BCD 码移到低 4 位
        MOV     SI,BX           ; 将 BX 中数移位后放回 SI 保存
        AND     BX,0FH          ; 清 BX 高 12 位
        ADD     AX,BX           ; AX 中数加 BX 中数作下一次的乘数
        POP     CX              ; CX 出栈
        LOOP    NEXT
        MOV     BX,AX           ; 将结果保存在 BX 中
        MOV     CL,16           ; 置(CL)=16,转换后的结果是 16 位的二进制数
SHOW:   MOV     DX,0            ; 逐位显示二进制的计算结果，将 DX 清 0
        SAL     BX,1            ; 算术左移 1 位
        RCL     DX,1            ; 带进位的循环左移 1 位
        ADD     DX,30H          ; 转换为对应的 ASCII 码
        MOV     AH,2            ; 显示
        INT     21H
        LOOP    SHOW
        MOV     AH,4CH
        INT     21H             ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（4）运行程序

运行结果如图 10.14 所示，压缩 BCD 码 1001001001100100 转换为二进制数为 10010000110000。

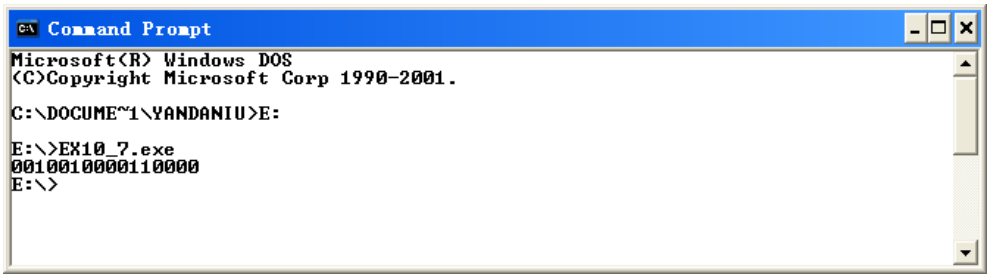


图 10.14　压缩 BCD 码转换二进制数结果

10.3.2　二进制数到其他进制数的转换

如将 16 位无符号的二进制数转换为 S 进制数（S 为 3～16 中的任一个整数），并且将 S 进制数的 ASCII 码按高位在前、低位在后（即高位在低地址，低位在高地址）的顺序，存放在由 SI 指示其起始偏移地址的字节存储区中。二进制数转换为 S 进制数往往采用除权值 S、取余、倒写的方法，其大致有如下三个步骤。

（1）将待转换的二进制数除以 S，得到第一个商数和第一个余数，这第一个余数就是所求的 S 进制的个位数。

（2）将第一个商数除以 S，得到第二个商数和余数，这第二个余数就是所求 S 进制数的十位数。

（3）跳转到步骤一循环这过程，直到商数为 0 时，所得到的余数就是所求 S 进制数的最高位数。

从上述“除 S 取余”过程可知，先得到的余数是 S 进制数的低位；后得到的余数是 S 进制数的高位。所以可利用堆栈“后进先出”的原则，将每次除 S 所得余数压栈保存。当商数为 0 时，再将保存在栈中的余数逐一弹出，将其转换成 ASCII 码后，送往 ASCII 码存储区。

1. 二进制数到十进制数的转换

【例 10.8】　编写程序，使其功能为：将十六位无符号二进制数转换为十进制数，并显示出来。

（1）分析问题

根据上面介绍可知，本题可在 BX 中存放权值除数 10，让(DX)(AX)逐次除以(BX)，余数在 DX 中。由于最后要按产生的余数从后到前显示。可以首先设计一个字节类型缓冲区 BUF，让 DI 指向 BUF，每次生成的余数加 30H 变成 ASCII 码后放到 DI 所指的存储区域，所有余数的最后加“$”，然后利用 9 号功能调用显示即可。

（2）流程图

流程图如图 10.15 所示。

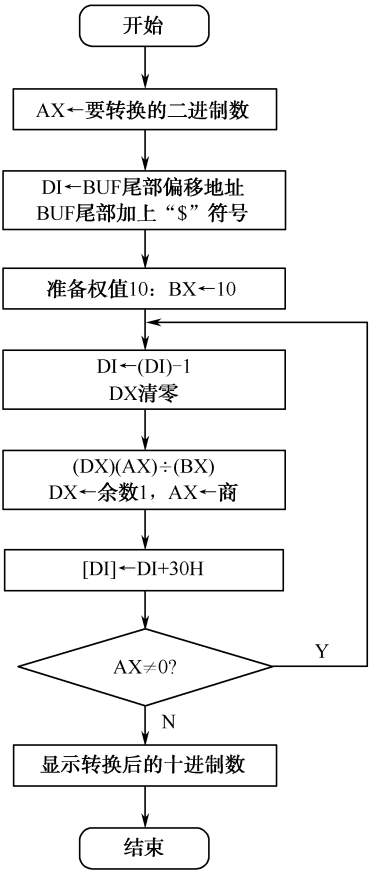


图 10.15　二进制数转换为十进制数并显示流程图

（3）编写程序

程序如下：

```
DATA    SEGMENT
BIN     DW  10010000110000B
BUF     DB   6 DUP(0)
DATA    ENDS
STACK1  SEGMENT      PARA     STACK
CODE    SEGMENT
        ASSUME  CS:CODE,DS,DATA,SS:STACK1
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AX,BIN              ; 将要计算的二进制数送入 AX 中
        LEA     DI,BUF              ; 指针 DI 指向 BUF 缓冲区
        ADD     DI,5                ; DI 指向 BUF 尾部
        MOV     BYTE PTR[DI],"$"    ; 所有余数的最后加"$"
        MOV     BX,10               ; 准备权值 10，作为除数
NEXT:   DEC     DI                  ; 指针向前移
        XOR     DX,DX               ; DX 清零，被除数已在 AX 中
```

```
        DIV     BX              ; 除以 10，余数在 DX 中，商在 AX 中，被除数 DX，AX
        ADD     DL,30H          ; 余数变为对应的 ASCII 码值
        MOV     BYTE PTR[DI],DL ; 保存转换后的十进制数结果
        CMP     AX,0            ; 判断商是否为 0
        JNZ     NEXT            ; 为 0 则转
        MOV     DX,DI           ; 显示转换后的十进制数，当前存储区首址放 DX
        MOV     AH,9            ; 调用 9 号功能调用显示
        INT     21H
        MOV     AH,4CH
        INT     21H             ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（4）运行程序

运行结果如图 10.16 所示，16 位二进制数 10010000110000 转换为十进制数为 9264。

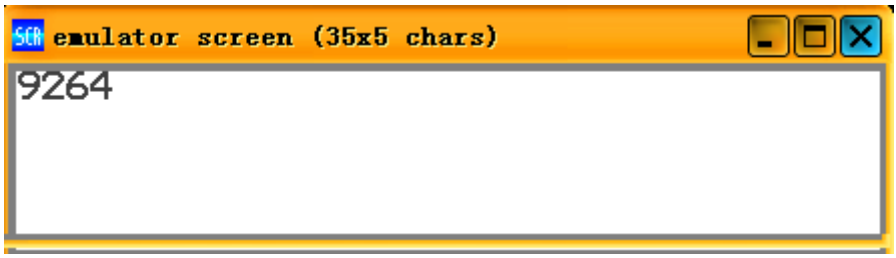


图 10.16　二进制数转换十进制数结果

说明：

例 10.8 中介绍了一种二进制数转换为十进制的方法。当然还可以根据给定的一个长度为 16 位的无符号二进制数，可以知道其表示的范围，测定此范围内含有多少个 10000（2710H）、1000（03E8H）、100（64H）、10（0AH），则可以分别获得其在万位、千位等的数目字，进而完成转换。若要显示出来，则可以把这些数及剩下的个位数拼上 30H，使之变成对应的 ASCII 码，一位一位地显示出来。采用此方法首先获得高位的数字。

2. 二进制数到十六进制数的转换

【例 10.9】　编写程序，使其功能为：将 16 位无符号二进制数转换为十六进制数，并显示出来。

（1）分析问题

本题和例 10.8 很相似，可在 BX 中存放权值除数 16，让(DX)(AX)逐次除以(BX)，余数在 DX 中。由于最后要按产生的余数从后到前显示。可以首先设计一个字节类型缓冲区 BUF，让 DI 指向 BUF，每次生成的余数加 30H 变成 ASCII 码后放到 DI 所指的存储区域，所有余数的最后加“$”，然后利用 9 号功能调用显示即可。

（2）编写程序

程序如下：

```
DATA    SEGMENT
BIN     DW  10010000110000B
BUF     DB  6 DUP(0)
```

```
DATA    ENDS
STACK1  SEGMENT     PARA    STACK
CODE    SEGMENT
        ASSUME  CS:CODE,DS,DATA,SS:STACK1
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AX,BIN                  ; 将要计算的二进制数送入 AX 中
        LEA     DI,BUF                  ; 指针 DI 指向 BUF 缓冲区
        ADD     DI,5                    ; 假定 BUF 从 0 起准备 6 个单元，DI 指向 BUF 尾
        MOV     BYTE PTR [DI],"$"       ; 所有余数的最后加"$"
        DEC     DI                      ; 指针向前移
        MOV     BYTE PTR[DI],"H"        ; 加十六进制单位 H
        MOV     BX,16                   ; 准备除以 16
CHANGE: DEC     DI                      ; 指针向前移
        MOV     DX,0                    ; 按题意二进制数在 DX 中，改存 AX 做除法
        DIV     BX                      ; (AX)除以 10 余数在 AH 中
        ADD     DL,30H                  ; 余数变为对应的 ASCII 码值
        CMP     DL,39H                  ; 是否是 A～F
        JBE     NEXT
        ADD     DL,7                    ; 如果是 A～F，再加 7
NEXT:   MOV     BYTE  PTR[DI],DL        ; 保存转换后的十六进制数结果
        CMP     AX,0                    ; 判断商是否为 0
        JNZ     CHANGE                  ; 不为 0 转 CHANGE
        MOV     DX,DI                   ; 显示转换后的十六进制数
        MOV     AH,9
        INT     21H
        MOV     AH,4CH
        INT     21H                     ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（3）运行程序

运行结果如图 10.17 所示，16 位二进制数 10010000110000 转换为十六进制数为 2430H。

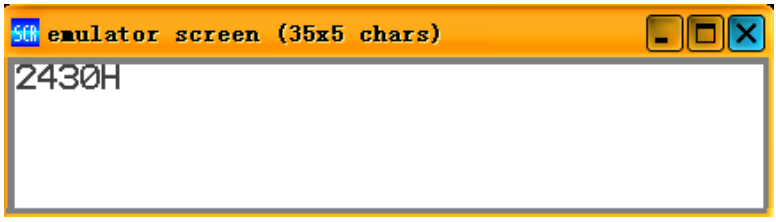


图 10.17　二进制数转换十六进制数结果

3. 二进制数到压缩 BCD 码的转换

【例 10.10】　编写程序，使其功能为：将 16 位无符号二进制数转换为压缩 BCD 码并显示出来。

（1）分析问题

本题涉及到两方面的内容，第一个方面内容是将 16 位无符号二进制数转换为压缩 BCD 码，第二方面内容是将压缩 BCD 码转化为 ASCII 码，并在屏幕上显示出来。

（2）编写程序

程序如下：

```
DATA    SEGMENT
BIN     DW  10010000110000B,0
DATA    ENDS
STACK1  SEGMENT     PARA     STACK
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AX,BIN          ; 将要计算的二进制数送入 AX 中
        MOV     SI,6            ; 假定缓冲区从 0 起准备 6 个单元，SI 指向缓冲区尾
        MOV     BX,10           ; 准备权值 10
CHANGE: DEC     SI              ; SI 减 1
        MOV     DX,0            ; 高 16 位 DX 清零，被除数已在 AX 中
        DIV     BX              ; (DX)(AX) 除以 10，第一个余数在 DX 中
        MOV     [SI],DL         ; 保存转换后的十进制数结果
        CMP     AX,0            ; 判断商是否为 0
        JZ      EXIT            ; 若为 0 则转 EXIT 退出
        XOR     DX,DX           ; DX 清零
        DIV     BX              ; (DX)(AX)除以 10，下一个余数在 DX 中
        MOV     CL,4
        SHL     DL,CL           ; DL 高位左移 4 位
        ADD     [SI],DL         ; 组合成一个压缩 BCD 码
        CMP     AX,0            ; 判断商是否为 0
        JNZ     CHANGE          ; 不为 0 转 CHANGE
EXIT:   MOV     AX,6            ; 寻求结果的长度
        SUB     AX,SI           ; 6 减目前 SI 的值为结果的长度
        MOV     CX,AX           ; 循环次数
        MOV     DH,0
SHOW:   PUSH    CX
        MOV     AL,[SI]         ; 取压缩 BCD 码
        INC     SI              ; 修改指针
        MOV     CH,AL
        MOV     CL,4
        SHR     AL,CL
        AND     AL,0FH          ; 取高半字节 BCD 码
        ADD     AL,30H          ; 形成对应的 ASCII 码
        MOV     DL,AL
        MOV     AH,2            ; 调用 2 号功能显示
        INT     21H
```

```
        MOV     AL,CH
        AND     AL,0FH          ; 取低半字节 BCD 码
        ADD     AL,30H          ; 形成对应的 ASCII 码
        MOV     DL,AL
        MOV     AH,2            ; 调用 2 号功能显示
        INT     21H
        POP     CX
        LOOP    SHOW
        MOV     DL,'H'           ; 显示 H
        MOV     AH,6
        INT     21H
        MOV     AH,4CH
        INT     21H             ; 返回 DOS 系统
CODE    ENDS
        END     START
```

（3）运行程序

运行结果如图 10.18 所示，16 位二进制数 10010000110000 转换为压缩 BCD 码为 9264H。

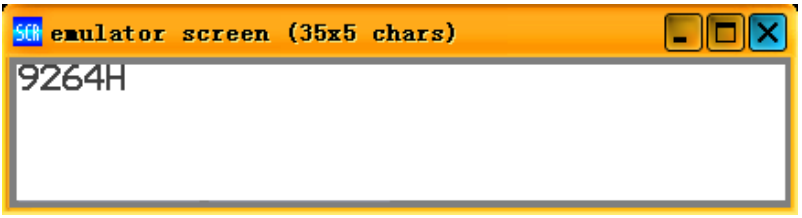


图 10.18　二进制数转换压缩 BCD 码结果

习题

1．请说明指令“LODSB”与如下程序片段的异同。

```
MOV     AL,[SI]
INC     SI
```

2．请说明指令“SCASB”与如下程序片段的异同。

```
CMP     AL,ES:[DI]
INC     DI
```

3．编写一个可以替代指令“REP MOVSW”的程序片段。

4．编写一个可以替代指令“REPNZ CMPSB”的程序片段。

5．编写一个子程序，实现字符串的逆转，如 ABCD => DCBA。

6．编写程序，使其功能是截取字符串某个子串。

7．编写程序，使其功能是把从键盘输入的一个小写字母用大写字母显示出来。

8．已知从内存 DATA 单元开始连续存放若干个字符，编写程序，使其功能是求出这些字符中出现次数最多的字符，并将出现次数及 ASCII 码分别存入 NUMB 和 DATAM 单

元。

9．编写一个程序，要求将字符串 2 插入字符串 1 的指定位置。

10. 编写一个简单的加密程序，将一个字符串中的所有字母字符按下列规律进行转换，非字母的字符保持不变。转换规律为'A'→'Z'，'B'→'Y'，……。

11．编写一段程序，接收 4 位十六进制数，然后用移位的方法把它转换成八进制数，并输出在屏幕上。

12．编写程序，实现从键盘输入任意一个字符，分别用二进制、八进制、十六进制、十进制输出其 ASCII 值。

13．从键盘输入一个表示年份的正整数(1～65535)，然后判断其是否为闰年（即被 100 整除也可被 400 整除的为闰年）。若是，则输出“Yes”；否则，输出“No”。

附录 A

ASCII 码字符表

ASCII 是 American Standard Code for Information Interchange 的缩写。它是国际标准化组织 ISO（International Organization for Standardization）批准的国际标准字符集 ISO 646 标准。

基本的 ASCII 字符集含有 128 个字符，其中 96 个为可打印字符，包括常用的字母、数字、标点符号等，另外还有 32 个控制字符。标准的 ASCII 码使用 7 位二进制数对字符进行编码。表 A.1 给出了基本 ASCII 码字符集及其编码。

表 A.1　ASCII 码字符表

十进制	十六进制	控制字符	十进制	十六进制	字符	十进制	十六进制	控制字符	十进制	十六进制	控制字符
0	00	NUL	17	11	DC1	34	22	″	51	33	3
1	01	SOH	18	12	DC2	35	23	#	52	34	4
2	02	STX	19	13	DC3	36	24	$	53	35	5
3	03	ETX	20	14	DC4	37	25	%	54	36	6
4	04	EOT	21	15	NAK	38	26	&	55	37	7
5	05	ENO	22	16	SYN	39	27	’	56	38	8
6	06	AQCK	23	17	ETB	40	28	(	57	39	9
7	07	Beep	24	18	CAN	41	29	)	58	3A	:
8	08	BSB	25	19	EM	42	2A	*	59	3B	;
9	09	Tab	26	1A	SUB	43	2B	+	60	3C	<
10	0A	LF	27	1B	ESC	44	2C	,	61	3D	=
11	0B	VT	28	1C	FS	45	2D	–	62	3E	>
12	0C	FF	29	1D	GS	46	2E	.	63	3F	?
13	0D	CF	30	1E	RS	47	2F	/	64	40	@
14	0E	SO	31	1F	US	48	30	0	65	41	A
15	0F	SI	32	20	SPACE	49	31	1	66	42	B
16	10	DLE	33	21	!	50	32	2	67	43	C

续表

十进制	十六进制	控制字符	十进制	十六进制	字符	十进制	十六进制	控制字符	十进制	十六进制	控制字符
68	44	D	83	53	S	98	62	b	113	71	q
69	45	E	84	54	T	99	63	c	114	72	r
70	46	F	85	55	U	100	64	d	115	73	s
71	47	G	86	56	V	101	65	e	116	74	t
72	48	H	87	57	W	102	66	f	117	75	u
73	49	I	88	58	X	103	67	g	118	76	v
74	4A	J	89	59	Y	104	68	h	119	77	w
75	4B	K	90	5A	Z	105	69	i	120	78	x
76	4C	L	91	5B	[	106	6A	j	121	79	y
77	4D	M	92	5C	\	107	6B	k	122	7A	z
78	4E	N	93	5D	]	108	6C	l	123	7B	{
79	4F	O	94	5E	^	109	6D	m	124	7C	|
80	50	P	95	5F	_	110	6E	n	125	7D	}
81	51	Q	96	60		111	6F	o	126	7E	/
82	52	R	97	61	a	112	70	p	127	7F	

注：NUL（NULL）为空字符，BEL（BELL）为响铃，LF（Line Feed）为换行，FF（Form Feed）为换页，CR（Carriage Return）为回车。

附录 B

DOS 系统功能调用（INT 21H）

DOS 功能调用，功能号在 AH 中，并设好其余的入口参数，向 DOS 发出 INT 21H 命令，最后获得出口参数。INT 21H 功能表如表 B.1 所示。

表 B.1　INT 21H 功能表

AH	功　能	入 口 参 数	出 口 参 数
00	终止程序	CS=程序段前缀	
01	键盘输入并回显		AL=输入字符
02	显示输出	DL=输出字符	
03	辅助设备（COM1）输入		AL=输入字符
04	辅助设备（COM1）输出	DL=输出字符	
05	打印机输出	DL=输出字符	
06	直接控制台 I/O	DL=FF（输入） DL=字符（输出）	AL=输入字符
07	键盘输入（无回显）		AL=输入字符
08	键盘输入（无回显） 检测 Ctrl+Break 或 Ctrl+C		AL=输入字符
09	显示字符串	DS:DX=串首址 字符串以“$”结尾	
0AH	键盘输入到缓冲区	DS:DX=缓冲区首址 (DS:DX)=缓冲区最大字符数	(DS:DX+1)=实际输入字符数
0BH	检测键盘状态		AL=00 有输入 AL=FF 无输入
0CH	清除缓冲区并 请求指定的输入功能	AL=输入功能号（1，6，7，8）	
0DH	磁盘复位		清除文件缓冲区
0EH	指定当前默认磁盘驱动器	DL=驱动器号 （0=A，1=B，2=C，…）	AL=系统中驱动器数

续表

AH	功　能	入口参数	出口参数
0FH	打开文件（FCB）	DS:DX=FCB 首地址	AL=00 文件找到 AL=FF 文件未找到
10H	关闭文件（FCB）	DS:DX=FCB 首地址	AL=00 目录修改成功 AL=FF 目录中未找到文件
11H	查找第一个目录项（FCB）	DS:DX=FCB 首地址	AL=00 找到匹配的目录项 AL=FF 未找到
12H	查找下一个目录项（FCB）	DS:DX=FCB 首地址 （使用通配符*或？）	AL=00 找到匹配的目录项 AL=FF 未找到
13H	删除文件（FCB）	DS:DX=FCB 首地址	AL=00 删除成功 AL=FF 文件未找到
14H	顺序读文件（FCB）	DS:DX=FCB 首地址	AL=00 读成功 =10 文件结束，未读到数据 =02DTA 边界错误 =03 文件结束，记录不完整
15H	顺序写文件(FCB)	DS:DX=FCB 首地址	AL=00 写成功 =01 磁盘满或是只读文件 =02 DTA 边界错误
16H	建立文件(FCB) 文件改名(FCB)	DS:DX=FCB 首地址 DS:DX=FCB 首地址 (DS:DX+1)=旧文件名 (DS:DX+17)=新文件名	AL=00 建立成功 =FF 无磁盘空间 AL=00 文件改名成功 =FF 文件改名失败
19H	取当前默认驱动器		AL=00 默认的驱动器号 0=A, 1=B, 2=C,…
1AH	设置 DTA 地址	DS:DX=DTA 地址	
1BH	取默认驱动器 FAT 信息		AL=每簇的扇区数 DS:BX=FAT 标识字节 CX=物理扇区的字节数 DX=默认驱动器的簇数
1CH	取指定驱动器 FAT 信息	DL=驱动器号	同上
1FH	取默认磁盘参数块		AL=00 无错 =FF 出错 DS:BX=磁盘参数块地址
21H	随机读文件(FCB)	DS:DX= FCB 首地址	AL=00 读成功 =01 文件结束 =02 DTA 边界错误 =03 读部分记录

续表

AH	功　能	入口参数	出口参数
22H	随机写文件(FCB)	DS:DX=FCB 首地址	AL=00 写成功 =01 磁盘满或是只读文件 =02 DTA 边界错误
23H	测定文件大小(FCB)	DS:DX=FCB 首地址	AL=00 成功，记录数填入 FCB =FF 未找到匹配的文件
24H	设置随机记录号	DS:DX=FCB 首地址	
25H	设置中断向量	DS:DX=中断向量 AL=中断类型号	
26H	建立程序段前缀	DX=新 PSP 段地址	
27H	随机分块读(FCB)	DS:DX=FCB 首地址 CX=记录数	AL=00 读成功 =01 文件结束 =02 DTA 边界错误 =03 读部分记录
28H	随机分块写(FCB)	DS:DX=FCB 首地址 CX=记录数	CX=读取的记录数 AL=00 写成功 =01 磁盘满或是只读文件 =02 DTA 边界错误
29H	分析文件名(FCB)	ES:DI= FCB 首址 DS:SI=字符串首址 AL=控制分析标志	AL=00 标准文件 =01 多义文件 =FF 非法盘符
2AH	取系统日期		CX=年(1980-2099) DH:DL=月：日 AL=星期(0～6)
2BH	设置系统日期	CX=年(1980-2099) DH:DL=月：日	AL=00 成功 =FF 无效
2CH	取系统时间		CH:CL=时：分 DH:DL=秒：1/100 秒
2DH	设置系统时间	CH:CL=时：分 DH:DL=秒：1/100 秒	AL=00 成功 =FF 无效
2EH	设置磁盘检验标志	AL=00 关闭标志 =01 打开标志	
2FH	取 DTA 地址		ES:BX= DTA 首地址
30H	取 DOS 版本号		AL=主版本号 AH=次版本号（发行号） BH=OEM 的序列号 BL:CX=24 位用户序列号

续表

AH	功　能	入口参数	出口参数
31H	结束并驻留	AL=返回码 DX=驻留区大小	
32H	取驱动器参数块	DL=驱动器号	AL=FF 驱动器无效 DS:BX=驱动器参数块地址
33H	CtrI+Break 检测	AL=00 取状态 AL=01 设置状态(DL) DL=00 关闭检测 DL=01 打开检测	取状态时： DL=00 关闭 Ctrl+Break 检测 =01 打开 Ctrl+Break 检测
35H	取中断向量	AL=中断类型	ES:BX=中断向量
36H	取空闲磁盘空间	DL=驱动器号 0=默认，1=A，2=B，…	成功：AX=每簇扇区数 BX=可用簇数 CX=每扇区字节数 DX=磁盘总簇数
38H	置/取国别信息	读国家信息： DS:DX=存放返回信息的地址 AL=00 取当前国别信息 AL=1～FF，BX=国家代码 设置国家信息： DX=FFFF AL=0～FF，BX=国家代码	成功：CF=0 BX=国家代码 （国际电话前缀码） 失败：CF=1 AX=错误代码
39H	建立子目录	DS:DX=ASCII Z 串地址	AX=错误码
3AH	删除子目录	DS:DX=ASCII Z 串地址	AX=错误码
3BH	改变当前目录	DS:DX=ASCII Z 串地址	AX=错误码
3CH	建立文件(handle)	DS:DX=ASCII Z 串地址 CX=文件属性	成功：AX=文件代号 失败：AX=错误码
3DH	打开文件(handle)	DS:DX=ASCII Z 串地址 AL=访问和文件共享方式 0=读，1=写，2=读/写	成功：AX=文件代号 失败：AX=错误码
3EH	关闭文件(handle)	BX=文件代号	失败：AX=错误码
3FH	读文件或设备(handle)	DS:DX=ASCII Z 串地址 BX=文件代号 CX=读取的字节数	成功：CF=0 AX=实际读入字节数 失败：CF=1，AX=错误码
40H	写文件或设备(handle)	DS:DX=ASCII Z 串地址 BX=文件代号 CX=写入的字节数	成功：CF=1 AX=实际写入字节数 失败：CF=1，AX=错误码

续表

AH	功　能	入口参数	出口参数
41H	删除文件	DS:DX=ASCII Z 串地址	成功：AX=00 失败：AX=错误码
42H	移动文件指针	BX=文件代号成功 CX:DX=位移量 AL=移动方式(0，1，2)	成功：DX:AX=新指针位置 失败：AX=错误码
43H	置/取文件属性	DS:DX=ASCII Z 串地址 AL=00 取文件属性 AL=01 置文件属性 CX=文件属性	成功：CX=文件属性 失败：AX=错误码
44H	设备文件 I/O 控制	BX=文件代号 AL=0 取设备信息 =1 置设备信息 =2 读字符设备 =3 写字符设备 =4 读块设备 =5 写块设备 =6 取输入状态 =7 取输出状态，… BL=驱动器代码 CX=读/写的字节数	成功：DX=设备信息 AX=传送的字节数 失败：AX=错误码
45H	复制文件代号	BX=文件代号 1	成功：AX=文件代号 2 失败：AX=错误码
46H	人工复制文件代号	BX=文件代号 1 CX=文件代号 2	失败：AX=错误码
47H	取当前目录路径名	DL=驱动器号 DS:SI=ASCII Z 串地址 （从根目录开始的路径名）	成功： DS:SI=当前 ASCII Z 串地址 失败：AX=错误码
48H	分配内存空间	BX=申请内存字节数	成功：AX=分配内存的首址 失败：AX=错误码 BX=最大可用空间
49H	释放已分配内存	ES=内存起始段地址	失败：AX=错误码
4AH	修改内存分配	ES=原内存起始段地址 BX=新申请内存字节数	失败：AX=错误码 BX=最大可用空间
4BH	装入/执行程序	DS:DX=ASCII Z 串地址 ES:BX=参数区首地址 AL=00 装入并执行程序 =03 装入程序，但不执行	成功：CF=0 失败：AX=错误码

续表

AH	功　能	入口参数	出口参数
4CH	带返回码终止	AL=返回码	
4DH	取返回代码		AX=返回代码 00=正常终止 01=用 Ctrl+C 终止 02=严重设备错误终止 03=用功能调用 31H 终止
4EH	查找第一个匹配文件	DS:DX=ASCII Z 串地址 CX=属性	失败：AX=错误码
4FH	查找下一个匹配文件	DS:DX=ASCII Z 串地址	失败：AX=错误码
50H	置 PSP 段地址	BX=新 PSP 段地址	
51H	取 PSP 段地址		BX=当前运行进程的 PSP
52H	取磁盘参数块		ES:BX=参数块链表指针
53H	BIOS 参数块(BPB)转换为 DOS 的驱动器参数块(DPB)	DS:SI=BPB 的指针 ES:BP= DPB 的指针	
54H	取盘的检验标志		AL=00 检验关闭 =01 检验打开
56H	文件改名	DS:DX=当前 ASCII Z 串地址 ES:DI=新 ASCII Z 串地址	失败：AX=错误码
57H	置/取文件日期和时间	BX=文件代号 AL=00 读取日期和时间 AL=01 设置日期和时间 (DX:CX)=日期：时间	失败：AX=错误码
58H	取/置内存分配策略码	取策略码：AL=00 置策略码：AL=01 BX=策略码	成功：AX=策略码 失败：AX=错误码
59H	取扩充错误码	BX=00	AX=扩充错误码 BH=错误类型 BL=建议的操作 CH=出错设备代码
5AH	建立临时文件	CX=文件属性 DS:DX=ASCIIZ 串地址	成功：AX=文件代号 DS:DX=ASCII Z 串地址 失败：AX=错误代码
5BH	建立新文件	CX=文件属性 DS:DX=ASCII Z 串地址	成功：AX=文件代号 失败：AX=错误代码

续表

AH	功　能	入口参数	出口参数
5CH	文件存取控制	AL=00 区域加锁 AL=01 区域解锁 BX=文件代号 CX:DX=区域偏移量 SI:DI=文件长度	成功：CF=0 失败：CF=1，AX=错误代码
5DH	取/置严重错误标志的地址	AL=06 取严重错误标志地址 AL=0A 置 ERROR 结构指针	DS:SI=严重错误标志的地址
60H	扩展为全路径名	DS:SI=ASCII Z 串的地址 ES:DI=工作缓冲区地址	失败：AX=错误代码
62H	取程序段前缀地址		BX=PSP 地址
68H	刷新缓冲区数据到磁盘	AL=文件代号	失败：AX=错误代码
6CH	扩充的文件打开/建立	AL=访问权限 BX=打开方式 CX=文件属性 DS:SI=ASCII Z 串地址	成功：AX=文件代号 CX=采取的动作 失败：AX=错误代码

注：AH=0～2EH 用于 DOSV1.0 以上；AH=2EH～57H 用于 DOSV2.0 以上；AH=58H～62H 用于 DOS3.0 以上；AH=63H～6CH 用于 DOSV4.0 以上。

表 B.2　AX 错误码表

错误码	错误类型	错误码	错误类型
01H	无效的功能号	0AH	不正确的环境
02H	文件未找到	0BH	不正确的格式
03H	路径未找到	0CH	无效的存取代码
04H	打开的文件太多	0DH	无效的数据
05H	拒绝存取	0EH	保留
06H	非法的文件号	0FH	指定的驱动器无效
07H	内存控制块破坏	10H	视图删除当前目录
08H	没有足够的内存空间	11H	非同一设备
09H	无效的内存块地址	12H	没有更多的文件

附录 C

BIOS 中断调用（INT n）

INT	AH	功　能	入口参数	出口参数
10H	00H	设置显示方式	AL=00 40×25 黑白文本，16 级灰度 =01 40×25　16 色文本 =02 80×25 黑白文本，16 级灰度 =03 80×25 16 色文本 =04 320×200 4 色图形 =05 320×200 黑白图形，4 级灰度 =06 640×200 黑白图形 =07 80×25 黑白文本 =08 160×200 16 色图形(MCGA) =09 320×200 16 色图形(MCGA) =0A 640×200 4 色图形(MCGA) =0D　320 × 200　16　色 图 形 (EGA/VGA) =0E 640×200 16色图形(EGA/VGA) =0F 640×350 单色图形(EGA/VGA) =10640×350 16 色图形(EGA/VGA) =11640×480 黑白图形(VGA) =12640×480 16 色图形(VGA) =13320×200 256 色图形(VGA)	
10H	01H	置光标类型	$(CH)_{0\sim3}$=光标起始行 $(CH)_{0\sim3}$=光标结束行	
10H	02H	置光标位置	BH=页号 DH/DL=行/列	
10H	03H	读光标位置	BH=页号	CH=光标起始行 CL=光标结束行 DH/DL=行/列

续表

INT	AH	功　能	入口参数	出口参数
10H	04H	读光笔位置		AX=0 光笔未触发 =1 光笔触发 CH/BX=像素行/列 DH/DL=字符行/列
10H	05H	置显示页	AL=页号	
10H	06H	屏幕初始化 或上卷	初始化窗口：AL=0 上卷：AL=上卷行数 BH=卷入行属性 CH/CL=左上角行/列号 DH/DL=右下角行/列号	
10H	07H	屏幕初始化 或下卷	初始化窗口：AL=0 下卷：AL=下卷行数 BH=卷入行属性 CH/CL=左上角行/列号 DH/DL=右下角行/列号	
10H	08H	读光标位置的字 符和属性	BH=显示页	AH/AL=字符/属性
10H	09H	在光标位置显示 字符和属性	BH=显示页 AL/BL=字符/属性 CX=字符重复次数	
10H	0AH	在光标位置显示字符	BH=显示页 AL=字符 CX=字符重复次数	
10H	0BH	置调色板 (320x200 图形)	BH=彩色调色板 ID BL=和 ID 配套使用的颜色	
10H	0CH	写像素	AL=颜色值 BH=页号 DX/CX=像素行/列	
10H	0DH	读像素	BH=页号 DX/CX=像素行/列	AL=像素的颜色值
10H	0EH	显示字符 （光标前移）	AL=字符 BH=页号 BL=前景色	
10H	0FH	取当前显示方式		BH=当前显示页页号 AH=字符列数 AL=显示模式

续表

INT	AH	功　能	入口参数	出口参数
10H	10H	置调色板寄存器 (EGA/VGA)	AL=0，1，2 置调色板，边框或寄存器 AL=7～1B 置各种读写或选择功能	
10H	11H	装入字符发生器 (EGANGA)	AL=0～4 全部或部分装入字符点阵集 AL=20H～24H 置图形方式显示字符集 AL=30H 读当前字符集信息	ES:BP=字符集位置
10H	12H	返回当前适配器设置的信 (EGA/VGA)	BL=10H（子功能）	BH=0 单色方式 =1 彩色方式 BL=VRAM 容量 (0=64K，1=128K，…) CH=特征位设置 CL=EGA 的开关设置
10H	13H	显示字符串	ES:BP=字符串地址 CX=字符串长度 AL=写方式(0～3) DH/DL=起始行/列 BH/BL=页号/属性	
11H		取设备信息		AX=返回值
12H		取内存容量		AX=字节数(KB)
13H	00H	磁盘系统复位	DL=驱动器号 （00～7FH:软盘，80H～0FFH:硬盘）	成功：AH=00 失败：AH=错误码
13H	01H	读磁盘驱动器状态	同上	AH=状态字节
13H	02H	读磁盘扇区	AL-扇区数 $(CL)_{6\sim7}(CH)_{0\sim7}$=磁道号 $(CL)_{0\sim5}$=扇区号 DH/DL=磁头号/驱动器号 ES:BX=数据缓冲区地址	读成功：AH=0 AL=读取的扇区数 读失败：AH=错误码
13H	03H	写磁盘扇区	同上	写成功：AH=0 AL=写入的扇区数 写失败：AH=错误码
13H	04H	检验磁盘扇区	AL=扇区数 $(CL)_{6\sim7}(CH)_{0\sim7}$=磁道号 $(CL)_{0\sim5}$=扇区号 DH/DL=磁头号/驱动器号	成功：AH=0 AL=检验的扇区数 失败：AH=错误码

续表

INT	AH	功　能	入口参数	出口参数
13H	05H	格式化磁道	AL=扇区数 $(CL)_{6\sim7}(CH)_{0\sim7}$=磁道号 $(CL)_{0\sim5}$=扇区号 DH/DL=磁头号/驱动器号 ES:BX=格式化参数表指针	成功：AH=0 失败：AH=错误码
14H	00H	初始化串行通信口	AL=初始化参数 DX=串口号	AH=串口状态 AL=调制解调器状态
14H	01H	写字符到串口	AL=字符 DX=串口号	写成功：$(AH)_7$=0 写失败：$(AH)_7$=1 $(AH)_{0\sim6}$=串口状态
14H	02H	从串口读字符	DX=串口号	读成功：$(AH)_7$=0 (AL)=字符 读失败：$(AH)_7$=1
14H	03H	取串口状态	DX=串口号	AH=串口状态 AL=调制解调器状态
14H	04H	初始化扩展 COM		
14H	05H	扩展 COM 控制		
15H	00H	启动盒式磁带机		
15H	01H	停止盒式磁带机		
15H	02H	磁带分块读	ES:BX=数据传输区地址 CX=字节数	AH=状态字节 =00 读成功 =01 冗余检验错 =02 无数据传输 =04 无引导 =80 非法字节
15H	03H	磁带分块写	DS:BX=数据传输区地址 CX=字节数	AH=状态字节 同上
16H	00H	从键盘读字符	AL=字符码	
16H	01H	取键盘缓冲区状态		AH=扫描码 ZF=0　AL=字符码 AH=扫描码 ZF=1 缓冲区无按键等待
16H	02H	取键盘标志字节		AL=键盘标志字节
17H	00H	打印字符 回送状态字节	AL=字符 DX=打印机号	AH=打印机状态字节

续表

INT	AH	功　能	入口参数	出口参数
17H	01H	初始化打印机 回送状态字节	DX=打印机号	AH=打印机状态字节
17H	02H	取打印机状态	DX=打印机号	AH=打印机状态字节
18H		ROM BASIC 语言		
19H		引导装入程序		
1AH	00H	读时钟		CH:CL=时:分 DH:DL=秒:1/100 秒
1AH	01H	置时钟	CH:CL=时：分 DH:DL=秒:1/100 秒	
1AH	02H	读实时钟		CH:CL=时:分(BCD) DH:DL=秒:1/100 秒(BCD)
1AH	06H	置报警时间	CH:CL=时:分(BCD) DH:DL=秒:1/100 秒(BCD)	
1AH	07H	清除报警		
33H	00H	鼠标复位	AL=00	BX=鼠标的键数
33H	00H	显示鼠标光标	AL=01	显示鼠标光标
33H	00H	隐藏鼠标光标	AL=02	隐藏鼠标光标
33H	00H	读鼠标状态	AL=03	BX=键状态 CX/DX=鼠标水平/垂直位置
33H	00H	显示鼠标位置	AL=04 CX/DX=鼠标水平/垂直位置	
33H	00H	设置图形光标	AL=09 BX/CX=鼠标水平/垂直中心 CS:DX=16×16 光标映像地址	安装了新的图形光标
33H	00H	设置文本光标	AL=0A BX=光标类型 CX=像素位掩码或起始的扫描线 DX=光标掩码或结束的扫描线	设置的文本光标
33H	00H	读移动计数器	AL=0B	CX/DX=鼠标水平/垂直距离
33H	00H	设置中断子程序	AL=0C　　CX=中断掩码 EX:DX=中断服务程序的地址	

附录 D

指令表

语 言 格 式	指 令 说 明	对标志位影响	意　　义
MOV dest, src	数据传送指令	无	src→dest
MOVSX dest, src	按符号扩展数据传送指令	无	src→dest．dest 左边位用 src 符号填，可对有符号数作符号扩展
MOVZX dest, src	按 0 扩展数据传送指令	无	src→dest，dest 左边位用 0 填，可对无符号数作 0 扩展
XCHG dest, src	数据交换指令	无	dest↔src
XLAT src/XLATB	查表转换指令	无	AL←(BX + AL)
LEA　dest,m	传送偏移地址指令	无	取 m 的偏移地址→(reg16)
LDS　dest,m	传送偏移地址和数据段首址命令	无	取 m 的偏移地址→(reg16)，取 m 的段地址→DS
LES　dest,m	传送偏移地址和附加段首址命令	无	取 m 的偏移地址→(reg16)，取 m 的段地址→ES
LFS　dest,src	传送偏移地址和附加段首址命令	无	取 src 的偏移地址→(reg16)，取 src 的段地址→FS
LGS　dest,src	传送偏移地址和附加段首址命令	无	取 src 的偏移地址→(reg16)，取 src 的段地址→GS
LSS　dest,src	传送偏移地址和堆栈段首址命令	无	取 src 的偏移地址→(reg16)，取 src 的段地址→SS
PUSH src	进栈指令	无	SP−1→SP，src 的高位→(SP)，SP−1→SP，src 的低位→(SP)
POP　dest	出栈指令	无	(SP)→dest 的低位，SP+l→SP． (SP)→dest 的高位，SP +1→SP
PUSHA	8 个 16 位通用寄存器入栈指令	无	AX, CX, DX, BX, SP/ESP, BP, SI, DI→(SP/ESP)；　SP/ESP －16 →SP/ESP

续表

语 言 格 式	指 令 说 明	对标志位影响	意 义
POPA	从堆栈弹出数据到 8 个 16 位通用寄存器指令	无	(SP/ESP)→DI, SI, BP, SP/ESP, BX, DX, CX, AX ; SP/ESP+16→SP/ESP
PUSHAD	8 个 32 位通用寄存器人栈指令	无	EAX, ECX, EDX, EBX, SP/ESP, EBP, ESI, EDI→(SP/ESP) ; SP/ESP - 32→SP/ESP
POPAD	从堆栈弹出数据到 8 个 32 位通用寄存器指令	无	(SP/ESP)→EDI, ESI, EBP, SP/ESP, EBX, EDX, ECX, EAX SP/ESP +32→SP/ESP
PUSHF	16 位标志人栈指令	无	SP−2→SP．标志寄存器内容→(SP)
POPF	从堆栈弹出 16 位标志指令	全按结果定	(SP)→标志寄存器，SP +2→SP
PUSHFD	32 位标志人栈指令	无	SP/ESP− 4→SP/ESP; EFLAGS→SP/ESP
POPFD	从堆栈弹出 32 位标志指令	全按结果定	SP/ESP→EFLAGS SP/ESP+4→SP/ESP
LAHF	标志送 AH 指令	无	取标志寄存器低 8 位→AH
SAHF	将 AH 送标志寄存器	CPAXS 按结果定	AH→标志寄存器的低 8 位
IN AL/AX/EAX, PORT	从端口 PORT 输入指令	无	(PORT)→AL/AX/EAX
OUT PORT, AL/AX/EAX	输出指令	无	AL/AX/EAX→PORT
INS dest, DX	输入字符串指令	无	DX 端口→ES：DI
INSB			DX 端口字节串→ES:DI
INSW			DX 端口字串→ES:DI
INSD			DX 端口双字串→ES:DI
OUTS DX, src	输出字符串指令	无	DS:SI→DX 端口
OUTSB			DS:SI 字节串→DX 端口
OUTSW			DS:SI 字串→DX 端口
OUTSD			DS:SI 双字串→DX 端口
ADD dest, src	加指令	CPAXSO 按结果定	dest←dest + src
ADC dest, src	带进位的加法指令	CPAXSO 按结果定	dest←dest + src + CF
INC dest	加 1 指令	CPAXSO 按结果定	dest←dest + 1
SUB dest, src	减指令	CPAXSO 按结果定	dest←dest − src
SBB dest, src	带借位的减法指令	CPAXSO 按结果定	dest←dest − src − CF
DEC dest	减 1 指令	CPAXSO 按结果定	dest←dest − 1
NEG dest	求补指令	CPAXSO 按结果定	dest←0 − dest
CMP dest, src	比较指令	CPAXSO 按结果定	dest − src

续表

语 言 格 式	指 令 说 明	对标志位影响	意　义
MUL src	无符号数乘法指令	CO 按结果定，PAXS 无定义	两个字节数相乘：AX←AL * src，两个字数据相乘：DX．AX←AX* src
IMUL src	有符号数乘法指令	CO 按结果定，PAXS 无定义	同上
DIV　src	无符号数除法指令	CPAXSO 无定义	AX/src:　AL←AX÷src 的商，AH←AX÷src 的余数，DX, AX/src: AX*-DX、AX÷src 的商，DX←DX、AX÷src 的余数
IDIV　src	有符号数除法指令	CPAXSO 无定义	同上
CBW	字节扩展为字指令	无	AL 为正，AH←00H；否则 AH←0FFH
CWD	字扩展为双字指令	无	AX 为正时 DX←0000H；否则 DX←0FFFFH
DAA	压缩 BCD 码加法调整	CPAXS 按结果定，O 无定义	压缩字节十进制加运算之后修正加法结果放在 AL 中
DAS	压缩 BCD 码减法调整	CPAXS 按结果定，O 无定义	压缩字节十进制加运算之后修正减法结果放在 AL 中
AAA	未压缩 BCD 码加法调整	CA 按结果定，ZSPO 无定义	非压缩字节十进制数 ASCII 码加运算之后修正加法结果放在 AL 中
AAS	未压缩 BCD 码加法调整	CA 按结果定，ZSPO 无定义	非压缩字节十进制数 ASCII 码减运算之后修正减法结果放在 AL 中
AAM	未压缩 BCD 码加法调整	CA 按结果定，ZSPO 无定义	非压缩字节十进制数 ASCII 码乘运算之后修正乘法结果放在 AX 中
AAD	未压缩 BCD 码加法调整	CA 按结果定，ZSPO 无定义	非压缩字节十进制数调节成压缩十进制数放入 AL 准备做除法
NOT dest	求反指令	无	dest←dest 的反
AND　dest, src	逻辑与指令	ZSP 按结果定，A 无定义，OC 置 0	dest←dest^src
OR　dest, src	逻辑或指令	ZSP 按结果定，A 无定义，OC 置 0	dest←dest^src
XOR　dest, src	逻辑异或指令	ZSP 按结果定，A 无定义，OC 置 0	dest←dest^src
TEST　dest, src	测试指令	ZSP 按结果定，A 无定义，OC 置 0	(OPNDl) ^ (OPND2)
SHL dest, src	逻辑左移	ZSPOC 按结果定，A 无定义	最高位→CF，dest←dest←dest* 2^{CL}
SAL　dest, src	算术左移	ZSPOC 按结果定，A 无定义	同上

续表

语 言 格 式	指 令 说 明	对标志位影响	意 义
SHR dest, src	逻辑右移	ZSPOC 按结果定，A 无定义	dest←dest÷2^{CL}，原最低位→CF
SAR dest. src	算术右移	ZSPOC 按结果定，A 无定义	dest←dest÷2^{CL}，原最低位→CF，最高位不变
ROL dest, src	循环左移	OC 按结果定	循环左移 CL 位，最高位→最低位，同时最高位→CF 标志
ROR dest, src	循环右移	OC 按结果定	循环右移 CL 位，最低位→最高位，最低位→CF
RCL dest, src	带进位循环左移	OC 按结果定	数据位、CF 一起看成环，循环左移 CL 位
RCR dest, src	带进位循环右移	OC 按结果定	数据位、CF 一起看成环，循环左移 CL 位
MOVS dest, src	串传送指令	无	根据类型(DS:SI/ESI)→(ES:DI/EDI)
MOVSB	字节串传送指令，与 REP/REPZ/REPNE 等联用，长度放 CX	无	(DS：SI/ESI)字节→(ES：DI/EDI)字节；DF=0 时，DI、SI 增；否则减
MOVSW	字节串传送指令，与 REP/REPZ/REPNE 等联用，长度放 CX	无	(DS：SI/ESI)字→(ES：DI/EDI)字；DF=0 时，DI、SI 增；否则减
MOVSD	串传送指令，与 REP/ REPZ/REPNE 等联用，长度放 cx	无	(DS：SI/ESI)双字→(ES：DI/EDI)双字
CMPS dest, src	串比较指令	ZSPOCA 按结果定	根据类型(DS：SI/ESI) -(ES：DI/EDI)
CMPSB	字节串比较指令，与 REP/REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(DS:SI)字节-(ES:DI)字节；DF=0 时，DI、SI 增；否则减
CMPSW	字串比较指令，与 REP/REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(DS: SI)字-(ES: DI)字；DF =O 时，DI、SI 增，否则减
CMPSD	串比较指令，与 REP/ REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(DS: SI/ESI)双字-(ES:DI/EDI) 双字；DF=0 时，DI、SI 增；否则减
SCAS dest	串搜索指令	ZSPOCA 按结果定	根据类型(AL/AX/EAX)-(ES:DI/EDI)
SCASB	串搜索指令，与 REP/ REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(AL/AX/EAX) -(ES：DI/EDI)字节；DF=0 时，DI、SI 增；否则减

续表

语 言 格 式	指 令 说 明	对标志位影响	意　义
SCASW	串搜索指令，与 REP/REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(AL/AX/EAX)-(ES:DI/EDI)字；DF=0 时，DI、SI 增，否则减
SCASD	串搜索指令，与 REP/REPZ/REPNE 等联用，长度放 CX	ZSPOCA 按结果定	(AL/AX/EAX)-(ES:DI/EDI)双字；DF=0 时，DI、SI 增；否则减
LODS　src	取字符串指令	无	(DS : SI)→(AL/AX/EAX)
LODSB	取字节字符串指令	无	(DS:SI)字节→(AL/AX/EAX)
LODSW	取字字符地串指令	无	(DS:SI)字→(AL/AX/EAX)
LODSD	取双字字符串指令	无	(DS:SI)双字→(AL/AX/EAX)
STOS　src	存字符串指令	无	(AL/AX/EAX)→(ES:DI)
STOSB	存字节字符串指令	无	(AL/AX/EAX)→(ES:DI)字节
STOSW	存字节字符串指令	无	(AL/AX/EAX)→(ES:DI)字
STOSD	存双字字符串指令	无	(AL/AX/EAX)→(ES:DI)双字
REP	重复前缀	无	宏、字串等处理重复 CX 次
REPE/REPZ	重复前缀	无	宏、字串等处理重复 CX 次，如 ZF=0 停止
REPNE/REPNZ	重复前缀	无	宏、字串等处理重复 CX 次，如 ZF =1 停止
JZ/JE 标号	等于 0/相等转移	无	判断 0 位 ZF 位为 1 时跳转到标号指定地址
JNZ/JNE 标号	不等于 0/不相等转移	无	判断 0 位 ZF 位为 0 时跳转到标号指定地址
JS 标号	为负转移	无	判断负位 SF 位为 l 时跳转到标号指定地址
JNS 标号	不为向转移	无	判断负位 SF 位为 0 时跳转到标号指定地址
JO 标号	溢出转移	无	溢出位 OF 位为 1 时跳转到标号指定地址
JNO 标号	未溢出转移	无	溢出位 OF 位为 0 时跳转到标号指定地址
JP/JPE 标号	奇校验转移	无	奇偶位 PF 位为 l 时跳转到标号指定地址
JNP/JPO 标号	偶校验转移	无	奇偶位 PF 位为 0 时跳转到标号指定地址
JC 标号	有进位转移	无	进位位 CF 位为 l 时跳转到标号指定地址
JNC 标号	无进位转移	无	进位位 CF 位为 0 时跳转到标号指定地址
JB/JNAE　标号	无符号数小于转移	无	CF=1，ZF =0 时跳转到标号指定地址
JNB/JAE　标号	无符号数不小于转移	无	CF=0，ZF =1 时跳转到标号指定地址
JA/JNBE 标号	无符号数大于转移	无	CF=0，ZF=0 时跳转到标号指定地址
JNA/JBE 标号	无符号数不大于转移	无	CF=l，ZF=1 时跳转到标号指定地址
JL/JNGE 标号	有符号数小于转移	无	SF≠OF∧ZF=0 时跳转到标号指定地址
JNL/JGE 标号	有符号数不小于转移	无	SF≠OF∨ZF =1 时跳转到标号指定地址
JG/JNLE 标号	有符号数大于转移	无	SF=OF∧ZF =0 时跳转到标号指定地址
JNG/JLE 标号	有符号数不大于转移	无	SF=OF∨ZF =1 时跳转到标号指定地址
JMP 标号	无条件转移	无	CX 变为 0 时跳转到标号指定地址

续表

语 言 格 式	指 令 说 明	对标志位影响	意 义
JCXZ 标号	CX 为 0 转移	无	CX←CX−1．若 CX≠0 则跳转标号指定地址
LOOP 标号	循环指令	无	CX←CX−1，若 CX≠0 且 ZF=1 则跳转
LOOPE/LOOPZ 标号	相等/为 0 循环指令	无	CX←CX−1，若 CX≠0 且 ZF=0 则跳转
LOOPNE/LOOPNZ 标号	不相等/不为 0 循环指令	无	IP←IP + DEST (或 EIP←EIP + DEST)
CALL DEST	调用子程序	无	CX 变为 0 时跳转到标号指定地址
RET [n]	从子程序返回	无	从堆栈栈顶弹出一个字送给 IP 或 EIP
IRET	中断返回	全按结果定	(SP)→lP,SP−2→SP,(SP)→CS,SP−2→SP，(SP)→标志寄存器。
CLC	清除进位标志	C 置 0	CF←0
STC	置进位标志	C 置 1	CF←1
CMC	进位标志取反	C 按结果定	CF←CF
CLD	清除方向标志	D 置 0	DF←0
STD	置方向标志	D 置 1	DF←1
CLI	清除中断标志	1 置 0	IF←0
STI	置中断标志	1 置 1	IF←1
NOP	空操作	无	不做任何操作“消耗”CPU 的时间
HLT	停机	无	暂停执行其后指令，等待中断服务程序返回
WAIT	等待	无	处于等待状态，等待外部设备“就绪”信号
ESC op，src	交权	无	协处理器取得其指令，并执行
LOCK：指令	总线封锁前缀	无	总线封锁直到与其联合的指令执行完
XADD dest，src	相加交换		TEMP←dest, dest← dest + src, src←(TEMP)

附录 E

伪指令表

语　句	伪指令名称
[变量名]DB/DW/DD/DQ/DT 表达式[，……]	定义字节 / 字 / 双字 / 4 字/10 字类型的数据存储区
记录名 RECORD 字段名：宽度[=表达式][，……]	定义记录类型（定义记录）
[变量名]记录名<表达式[，……]>	定义记录类型数据区（记录变量）
结构名 STRUCT 　　数据定义语句序列 结构名 ENDS	定义结构类型（定义结构）
结构变量名结构名<字段值表>	定义结构变量（定义结构类型数据区）
符号名 EQU 表达式	等价，定义常量
符号名=表达式	等号，定义常量
变量名 / 标号 LABEL 类型	定义符号名
段名 SEGMENT[使用类型][边界类型][连接类型]['类别']	段定义
段名 ENDS	段结束
ASSUME 段寄存器：段名[，……]	假定默认段
组名 GROUP 段名[，……]	定义组
过程名 PROC [NEAR] /FAR	定义过程
过程名 ENDP	过程结束
ORG 数值表达式	置汇编地址计数器，以下代码在该处开始存放
NAME 模块名	定义模块名
END[表达式]	模块结束
TITLE 正文	输出标题
SUBTTL 正文	输出子标题
PUBLIC 符号[，……]	说明公共符号
EXTRN 符号：类型[，……]	说明外部符号
宏指令名 MACRO 形参[，……]	宏定义
ENDM	宏定义结束
LOCAL 形参[，……]	产生特殊符号

续表

语　句	伪指令名称
PURGE 宏指令名[，……]	取消宏定义
EXITM	终止宏扩展
REPT 表达式	固定重复
IRP 形参，<实参[，……]>	不定重复汇编
IRPC 形参，字符串	单字符不定重复
IF 数值表达式 IFE 数值表达式	测试表达式
IFDEF 符号 IFNDEF 符号	测试符号是否定义
IF1 IF2	测试扫描次数
IFB<参数> IFNB<参数>	测试参数是否为空
IFIDN<参数 I>，<参数 2> IFNIDN<参数 1>，<参数 2>	测试字符串是否相等
ENDIF	条件汇编结束
. LIST . XLIST	汇编列表文件输出控制
. LALL . XALL . SALL	宏展开列表输出控制
. CREF . XCREF	符号交叉引用文件输出控制
INCLUDE 文本文件名	将指定文本加入源文件